凝聚名家技术典范 · 分享成功IT之路



DB2数据库性能调整和优化(第3版)



牛新庄 著

清华大学出版社

DB2 数据库性能调整和优化

(第 3 版)

牛新庄　著

清华大学出版社

北　　京

内 容 简 介

本书侧重于介绍DB2数据库的性能调优。性能调优是一个系统工程：全面监控分析操作系统、I/O性能、内存、应用及数据库才能快速找到问题根源；深刻理解DB2的锁及并发机制、索引原理、数据库参数、优化器原理、统计分析和碎片整理等内部机理才能针对性地快速提出解决问题的方法；快照、db2pd、db2expln以及各种管理视图和表函数等则是必须熟练掌握的工具。本书覆盖了进行DB2数据库性能调优所需的全部知识和工具，并提供了大量的性能调优的实际案例，这些案例都基于作者10多年积累的经验和总结，其中包括了近年来大型银行系统实际遇到的案例。本书还首次涵盖了针对DB2 pureScale及同城双活GDPC(地理上分离的pureScale集群)的性能调优方法和实践。

本书适合有一定DB2数据库基础知识和经验的数据库工程师，以及希望深入、全面地掌握DB2数据库性能分析和调优知识的读者，同时可以成为数据库软件开发人员开发高性能数据库软件的参考书。

图书在版编目(CIP)数据

DB2数据库性能调整和优化 / 牛新庄 著. —3版. —北京：清华大学出版社，2017
ISBN 978-7-302-48118-8

Ⅰ. ①D…　Ⅱ. ①牛…　Ⅲ. ①关系数据库系统　Ⅳ. ①TP311.132.3

中国版本图书馆CIP数据核字(2017)第208131号

责任编辑： 王　军　李维杰
装帧设计： 牛艳敏
责任校对： 成凤进
责任印制：

出版发行： 清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　**邮　　编：** 100084
社 总 机：010-62770175　**邮　　购：** 010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销： 全国新华书店
开　　本： 185mm×230mm　**印　张：** 33.5　**字　数：** 690千字
版　　次： 2009年5月第1版　2017年9月第3版　**印　　次：** 2017年9月第1次印刷
印　　数： 1~4000
定　　价： 98.00元

产品编号：

序

关系型数据库已经走过整整半个世纪曲折而辉煌的历程，回顾 50 年的关系数据库发展史，我们心潮涌动，激情难抑。关系数据库始于 1970 年 IBM 公司研究员 E.F.Codd 博士，即“关系数据库之父”，发表的业界第一篇关于关系数据库理论的论文“A Relational Model of Data for Large Shared Data Banks”。从此关系型数据库如雨后春笋般四处萌发，各大厂商争先恐后，加入到关系型数据库的研发大潮中，而后又如大浪淘沙般去璞存真，时至今日，留下的是真正适用于客户、适应于潮流的关系型数据库产品。

IBM 公司作为关系数据库的推广先锋，为业界提供了一批优秀的数据库技术领域先驱科学家，他们所研发出的 DB2 数据库，经过近 50 年的发展，已经广泛应用于金融、电信、制造等多个行业，对日常的工作和生活带来了深远的影响。

在中国，DB2 数据库的兴起大概在 15 年前，当时国内鲜有应用，牛新庄先生作为国内首批接触研究 DB2 数据库的工程师，为业界提供了大量技术服务和专业培训，为 DB2 数据库的推广应用做出了积极杰出的贡献。同时，也为 IBM 公司反馈了很多很好的 DB2 研发建议，为关系型数据技术的长远发展贡献智慧。在数据库方向的精深造诣和丰富实践，就浓缩在他的 DB2 数据库著作中，为广大 IT 同仁授业解惑。

本套图书可以说是伴随着 DB2 数据库的成长，从第一版主讲 DB2 V8，到第二版的 DB2 V9，再到第三版的 DB2 V10。基本上每出一个版本我都会仔细品味，每个版本作者都很用心，都会删减对当前不适用的章节，加入很多新的功能和他近期实际经历的案例。经典性、权威性、实用性是本套书籍锁定的主要目标。

第一，本套书籍涵盖了 DB2 的几乎所有功能，是业界最大规模的系统梳理与总结。从理论知识到最佳实践，无所不包，无所不有，同时还总结了几十个最佳案例。

第二，本套书籍理论讲解深入浅出，案例多样详实，无论是对零基础还是拥有多年 DBA 工作经验的人都非常适用。

第三，精品的价值在于传世久远，经典的意义在于常读常新。我认为，只有被广泛阅读，受到大家喜爱、接受的作品，才具有经典的资质与意义。希望作者继续努力，将本套书籍打磨成关系数据库的经典图书。

其中第一本书《循序渐进 DB2　DBA 系统管理、运维与应用案例(第 3 版)》，是 DB2 学习的入门书籍。该书包含了从入门到中级阶段的知识和技能的介绍，全面展示了 DB2 的主要功能和日常的工作技巧，尤其是实例和案例部分，这部分内容从实际出发为读者列举了常见的案例场景和处理办法，非常实用。在新的一版中，所有的内容、示例都基于 DB2 V10.5 版本进行了修订，并介绍了 DB2 V11.1 中的新功能、新特性。

第二本书《DB2 高级管理、系统设计与诊断案例(第 3 版)》是 DB2 学习的高级进阶，该书从 DB2 体系结构入手，介绍了 DB2 各个内部组件的层次与功能、内存内部结构、存储内部结构、锁和并发原理等。在理解 DB2 内部基本原理的基础上进一步介绍了 DB2 的高级功能，包括分区功能、高级压缩功能等。此外，系统介绍了 OLTP 和 OLAP 系统的设计方法和管理技术、高可用和容灾方案，以及集群技术，其中涉及 HADR、DPF 和 pureScale 技术，以及地理上分离的 pureScale 集群(GDPC)技术。该书还介绍了 DB2 各种监控和诊断方法，通过精选的诊断案例使读者在学习知识的同时积累实践经验。

第三本书《DB2 数据库性能调整和优化(第 3 版)》专门介绍 DB2 性能调整和优化，从 DB2 数据库性能有关的基础知识和原理入手，从数据库所处的运行环境(OS、存储等)开始介绍，并对 DB2 的进程和内存进行深入讲解。在全面了解性能相关知识后，开始逐步展开，从设计到监控，从配置参数到调优工具，从锁和并发到优化器统计信息，最后列出了几个完整的性能调优案例以增加技术理解。

祝愿每一位读者能有所得、有所悟，成长为新一代的数据技术专家，也祝愿牛新庄先生在数据技术领域这条康庄大道上走得更宽更远。

IBM 前大中华区总经理　　王天義
IBM 大中华区高级顾问

前　言

自 1999 年左右我开始从事数据库有关的技术工作到现在已近 20 年时间，此期间信息科技飞速发展，从无纸化办公和数据大集中到移动互联和大数据、人工智能、云计算等信息技术改变了生活，并颠覆了传统商业模式。信息科技的发展离不开数据处理技术的进步，在这一轮信息化浪潮中，数据处理技术也发生了翻天覆地的变化，对企业经营发展和对外服务的意义越来越重要。一方面，传统企业级数据库的能力，在原有的道路和方向上不断地持续提升演进，以满足企业市场不断迸发的各类需求。另一方面，互联网场景孕育的各种新兴的数据处理技术亦不断涌现，例如 NewSQL、NoSQL、Hadoop 等大数据处理技术，这些技术成为传统数据库产品的必然补充，同时也对传统数据库产品产生了一定的冲击。但是以我长期从事企业数据处理相关工作的经验看，在企业级市场尤其是金融企业市场里面，传统数据库产品的能力依然是解决企业主要业务需求的不二选择。因此，传统数据库技术的研究和应用仍然是信息科技工作的重点。

近年来传统数据库产品在不断改进升级，以支持更快的处理能力和更高的可用性，满足不同场景下的用户需求。DB2 作为一款主流数据库产品，在这些方面也都进步明显，例如 Purescale 集群技术、跨数据中心的 GDPC 技术、列存储的 BLU 技术等创新功能就表现不俗，满足了特定业务场景需求，给企业带来了很大的价值提升。特别是 GDPC 技术，帮助企业搭建关键业务系统同城对等全双活生产架构，为最终用户提供高等级容灾的连续服务，对企业对外服务的提升意义非凡，也使数据库从业者们领略了 DB2 产品创新的精华。

基于 DB2 产品的演进以及近些年的思考和实践，我重新梳理了之前编写的第 2 版的 3 本 DB2 系列技术图书，对其进行了大篇幅的修改和重写，力图对近些年实践的精华和 DB2

产品的新趋势进行总结。在此奉献给各位数据库从业的同仁，在技术的路上共勉。

由于本人水平有限，时间有限，书中不免有这样或者那样的错误，希望广大读者朋友不吝赐教指正！

最后，感谢我的家人和同事在本书重写过程中的帮助，谢谢你们！

牛新庄

目　录

第 1 章 性能调整概述

为什么要进行性能调优呢？因为我们的应用系统在运行一段时间后，用户报告系统运行会变慢，使他们不能完成所有的工作，完成事务和处理查询花费了过长的时间，或者应用程序在一天中的某些时段变慢。要确定造成问题的本质原因，必须评估系统资源的实际使用情况并进一步地分析资源使用的瓶颈所在。

用户通常报告以下性能问题：

- 事务或查询的响应时间比预期的长
- 事务吞吐量不足以完成必需的工作负载
- 事务吞吐量减少

为了维持数据库应用程序的最优性能，应该制定一个计划用于评估系统性能，以便在性能出现问题时，该计划可以根据性能问题的情况对数据库做出调整，以维持良好的性能。定期的、特定的评估能够帮助您预见并纠正性能问题。通过尽早识别出问题，可以有效地防止这些问题严重地影响用户。

本章我们将通过一个实际生产中的客户案例，展开对性能调整的讲解。本章主要内容包括：

- 性能概述
- 性能目标
- 性能评估
- 性能模型

- 什么时候需要性能调整
- 性能调整准则
- 性能调整的方法和过程
- 保持良好性能

1.1　性能概述

首先让我们以一个真实银行的应用案例来展开性能调整的话题，图 1-1 和图 1-2 分别为该行手机银行系统的逻辑架构部署图和物理架构部署图。

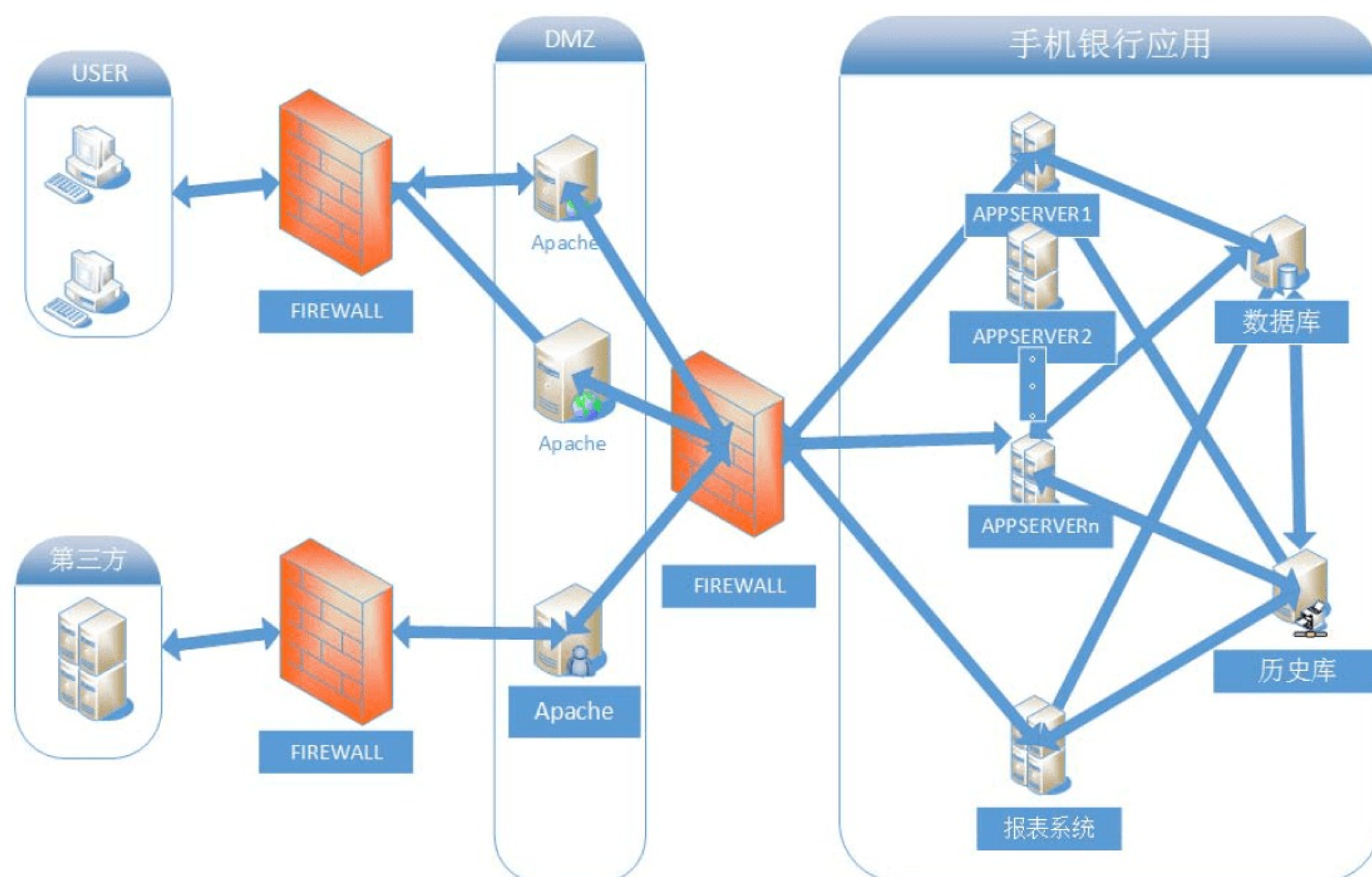


图 1-1　某手机银行系统的逻辑结构图

图 1-2 中的手机银行系统是一个大型的复杂系统。在这个系统中，从上至下包括以下几个层次：应用程序、中间件应用服务器、数据库、主机系统(操作系统)、光纤交换机和 SAN 存储网络(EMC VMAX)。在系统发生性能问题时，性能问题的定位和调优很复杂。

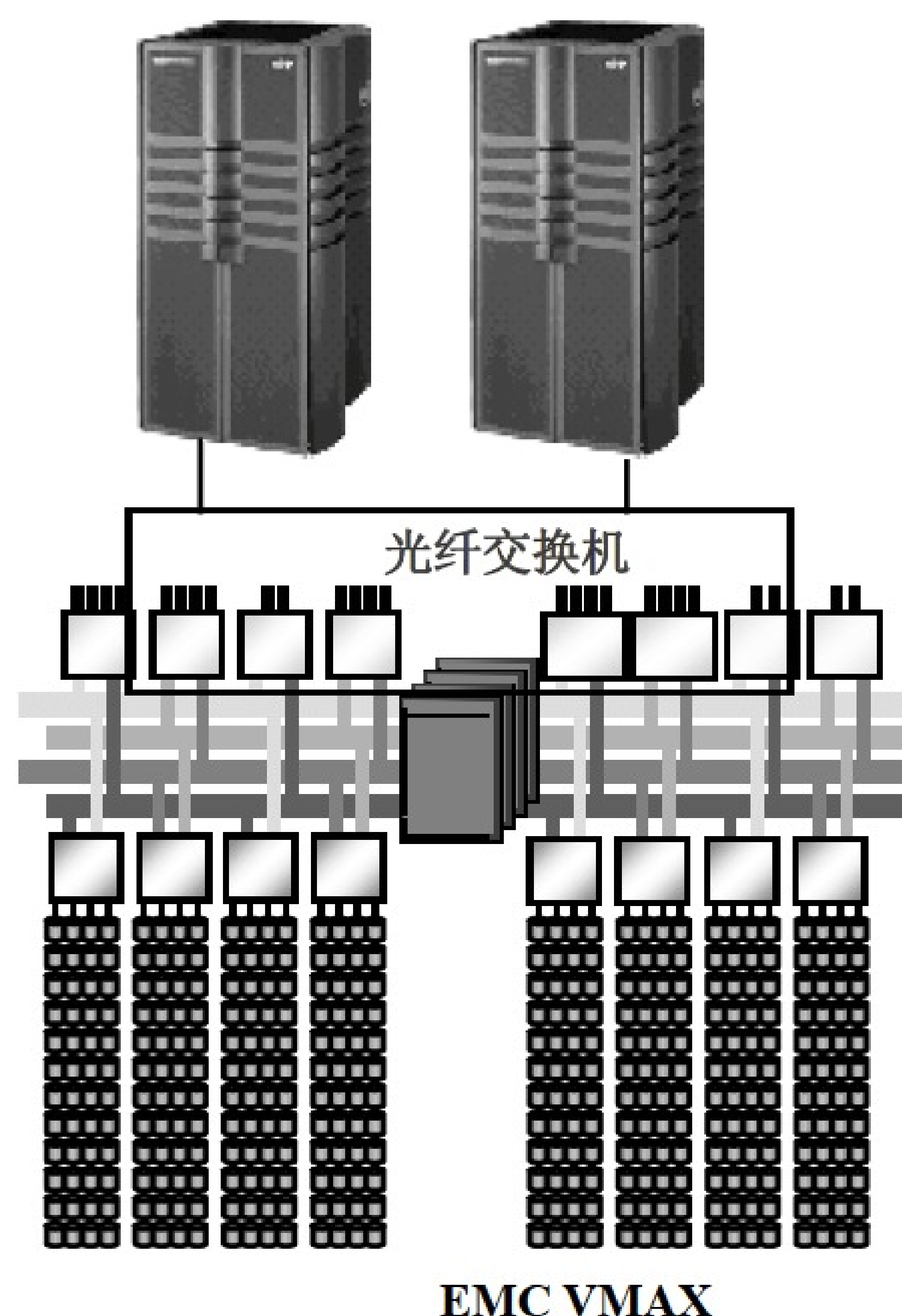


图 1-2　某手机银行系统的物理结构图

该手机银行系统的信息架构如下：

- 存储：采用的是 EMC VMAX 存储系统
- 操作系统：IBM AIX 7.1 (p750 lpar)
- 数据库：IBM DB2 9.7
- 中间件：Oracle Weblogic
- Web 服务器：Linux Apache Server
- 应用：采用基于 J2EE 的 Java 应用

该系统自从 2013 年上线以来，业务量逐年增长，运行至今，对最终用户的交互响应速度不如预期。在主机系统上观察，CPU 较忙，在业务繁忙时一直保持在 50%左右。手机银行系统由业务应用程序、DB2 数据库、AIX 主机、EMC 存储多个部分组成，因此性能瓶颈的定位和性能的优化都比较复杂。

那么我们如何来解决这些性能问题呢？有两种方法：

- 第一种方法是我们可以通过扩容硬件物理资源(增加 CPU、内存以及购买更快的存储系统)来实现。
- 第二种方法是我们试图对应用系统做出相应的调整来优化系统以改善目前的情况。

第一种方法我们需要投入更多的经济成本，而第二种方法需要我们利用经验来对整个系统做出调整。本书中要介绍的是第二种方法。

在进行调整之前，您有必要先了解关于性能调优方面的某些话题。

首先，性能的概念是什么呢？性能是业务应用系统(例如我们本章所举的手机银行系统案例)在特定硬件资源(例如 32 路 CPU、128GB 内存)和工作负载下所表现出来的处理能力。性能主要通过系统响应时间、吞吐量和可用性来衡量。

性能受以下因素影响：

- 系统中可用的物理资源
- 如何充分合理地利用这些资源

一般情况下，通过性能调整我们可以完成以下目标：

- 处理更大的或更紧迫的工作负载，而不增加处理成本，例如增加工作负载而不用购买新硬件或占用更多处理器时间
- 获得更快的系统响应时间或更大的吞吐量，而不增加处理成本
- 降低处理成本，而不会降低对用户的服务

1.2 性能评估

以下评估描述了事务处理系统的性能：

- 吞吐量
- 响应时间
- 每个事务的成本
- 资源利用率

1. 吞吐量

吞吐量用于评估系统的整体性能。对于事务处理系统，吞吐量通常用每秒事务数(TPS)或每分钟事务数(TPM)来计量。吞吐量取决于以下因素：

- 服务器硬件资源配置
- 软件中的处理开销
- 磁盘上数据的布局
- 硬件和软件都支持的并行度
- 正在处理的事务类型

2. 响应时间

响应时间用于评估单个事务或查询的性能。通常认为，响应时间是从用户输入一个命令或激活一个功能开始一直到应用程序指示已完成该命令或功能所消耗的时间。典型 DB2 应用程序的响应时间包括以下操作序列(以查询为例，每个操作都需要一定的时间，响应时间不包括用户思考和输入查询或请求的时间)：

(1) 应用程序将查询请求转发到数据库服务器。

(2) 数据库服务器执行查询优化并检索所有用户定义的例程，生成该查询对应的执行计划。

(3) 数据库服务器按照执行计划检索出适当的记录，如果该记录当前不在内存中，则还需要磁盘 I/O 操作。

(4) 数据库服务器在执行查询的过程中，有可能还会执行一些后台操作，这些操作也有可能影响查询的响应时间(比如记录日志和清除赃页面)。

(5) 数据库服务器将结果返回给应用程序。

(6) 应用程序将数据库结果返回给用户，等待用户的进一步操作。

图 1-3 显示了步骤(1)~(6)中所述的操作如何作用于整体响应时间。

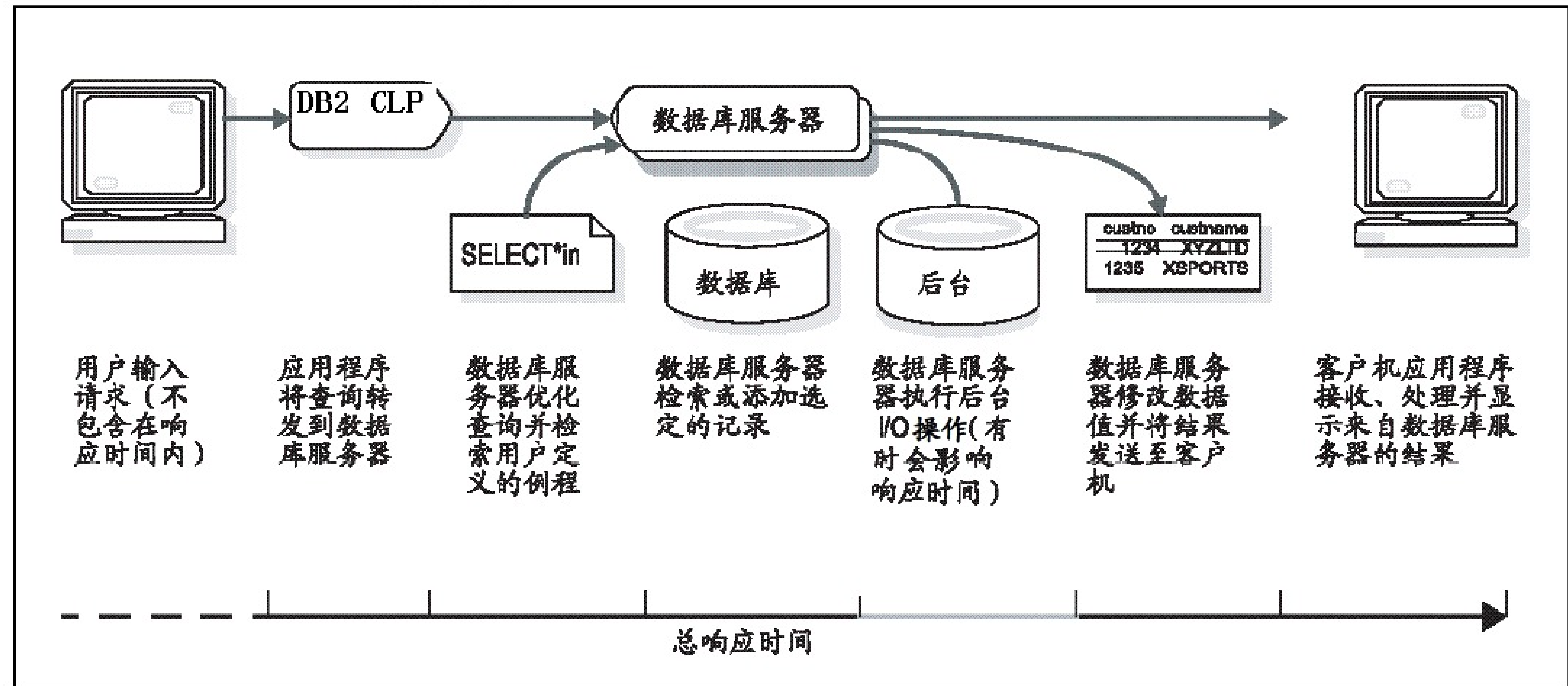


图 1-3　事务的响应时间

3. 响应时间和吞吐量

响应时间和吞吐量是相关联的。在您增加总体吞吐量时一般单个事务的响应时间会增加，因此您必须权衡吞吐量和响应时间的关系。您可以通过为特定查询分配不成比例的资源数量，在牺牲总体吞吐量的情况下减少该查询的响应时间。与之相反，可以通过限制数

据库分配给大型查询的资源数来维持总体吞吐量。

当尝试在对高事务吞吐量的当前需求和对执行大型决策支持查询的即时需求之间取得平衡时，吞吐量与响应时间之间的平衡就变得明显起来。应用于查询的资源越多，可用于处理事务的资源就越少，并且查询对事务吞吐量的影响就越大。相反，提供给查询的资源越少，查询花费的时间就越长。

4. 每个事务的成本

每个事务的成本是财务上的量度，通常用于比较应用程序、数据库服务器或硬件平台之间的总体操作成本。

要评估每个事务的成本，通常采用下面的方法：

(1) 计算与运行应用程序相关的所有成本，这些成本可能包括硬件和软件的安装成本、运作成本及其他费用。

(2) 设计应用程序有效期的事务和查询的总数。

(3) 用总成本除以事务总数。

尽管该测量对于进行规划和评估很有用，但是它与达到最佳性能的运行问题几乎无关。

5. 资源利用率和性能

资源利用率与负载类型和并发性有很大关系，一天、一周、一月、一年中的峰值负载以及决策支持(Decision Support System，DSS)查询或备份操作所施加的负载对于任何容量将用尽的系统都会产生明显的影响。比如银行的核心系统，既要满足每天的存取款处理，又要满足各种批量程序的大并发处理，如果您只考虑了每天的存取款处理而忽略了大并发的批量处理，那么分配过低的系统资源无疑会对该系统造成毁灭性的影响。

因此在系统上线之前和上线之后必须对系统的工作负载和性能进行定期评估，以预测峰值负载并比较使用周期中不同时刻的性能指标，通过趋势分析来确定未来的容量需求。定期评估有助于为数据库服务器上的应用程序开发总体的性能概要文件，该概要文件对于确定如何可靠地提高性能具有关键意义。

关于操作系统提供的用于评估对系统和硬件资源的性能影响的工具，请参阅本书第 2 章“操作系统及存储的性能调优”。

资源利用率是与系统资源可用的总时间相比，该系统资源实际被占用的时间的百分比。例如，如果 CPU 在一分钟内总共用 42 秒处理事务，那么在这段时间间隔内的利用率就是 70%。

定期评估并记录以下系统资源的利用率：

- CPU
- 内存
- 磁盘
- 网络

当某项资源被过度使用或者它的利用率与其他系统资源的利用率不成比例时，就称该资源对于性能是临界的。例如，当一个磁盘的繁忙率达到 70%，而系统中其他所有磁盘的繁忙率只有 30%的时候，可以认为该磁盘是临界的或是被过度使用的。尽管 70%并不表示磁盘被严重过度使用，但可以通过重新安排数据来平衡整个磁盘集上的 I/O 请求，从而提高系统的性能。

如何评估资源利用率取决于操作系统为报告系统活动和资源利用率所提供的工具。一旦发现看起来被过度使用的资源，就可以使用数据库服务器的性能监视实用程序来收集数据，并对可能占用该系统资源上负载的数据库活动进行干涉。可以调整数据库的配置参数或操作系统的相关 I/O 配置，以减少那些数据库活动或将它们分散到其他资源中(关于这部分的详细内容请参见本书第 2 章)。

1.3　建立性能目标

性能调整的目标是提高系统的吞吐量，缩短系统的响应时间，支持众多的并发用户，缩短数据加载的时间。

无论是正在设计还是在维护系统，您都应该建立专门的性能目标，它使您知道何时要做调整。如果您试图胡乱地改动配置参数或 SQL 语句，就可能会浪费调整系统的时间，甚至会使性能变得更糟糕。

为数据库服务器及其支持的应用程序建立性能目标要考虑许多注意事项。建立性能目标时，请考虑以下问题：

- 您最关注的是什么？是最大化事务吞吐量、最小化特定查询的响应时间还是实现两者的混合？
- 在简单的事务(OLTP)请求、复杂的决策支持查询(DSS)请求和其他类型(批处理、报表等)请求之间，数据库服务器通常处理什么类型的请求或混合的请求？
- 您需要高性能还是高可用性？有时候为了达到高性能就需要承受数据丢失的风险。在什么时候您希望用事务处理速度(性能)与可用性或丢失特殊事务的风险相交换？
- 您期望的最大并发用户数和最大事务量是多少？
- 您的配置受内存、磁盘空间或 CPU 等资源的限制吗？

这些问题的回答能够帮助您为资源和应用程序设置合理的现实的性能目标。

1.4 什么时候需要做性能调整

多数人认为只有当用户感觉性能差时才进行调整，此时即使使用调整过程中最有效的调整策略，往往也太迟了。此时，如果你不愿意重新设计应用的话，就只能通过重新分配内存或调整 I/O 的办法或多或少地提高性能。DB2 提供了许多特性，这些特性只有应用到正确设计的系统中时才能够很大地提高性能。

其实，性能调整应该跨越整个应用系统的开发生命周期。不光是 DBA 用户，所有系统用户都会以某种方式对性能产生影响，这包括数据库服务器管理员、数据库管理员、应用程序设计人员和客户机应用程序用户。

数据库服务器管理员通常要协调所有用户的活动，以确保系统性能达到总体预期效果。例如，操作系统管理员可能需要重新配置操作系统以增加共享内存量，或关闭操作系统以安装新的配置，这需要关闭数据库服务器。此时，数据库服务器管理员必须对这个停机时间做出安排，并向所有受影响的用户通知系统将在何时不可用。

系统管理员应该：

- 了解发生的所有与性能相关的活动，使用户了解性能的重要性，与性能相关的活动是如何影响他们的，以及如何协助以获得并保持最佳性能。

数据库管理员应该注意：

- 表和查询如何影响数据库服务器的总体性能。
- 表和表分区的位置。
- 创建最合理的索引和配置参数。
- 观察 SQL 的执行计划是否最优，统计分析是否已经过时。
- 数据在磁盘上的分布是如何影响性能的。
- 存储的 I/O 设计，数据库的物理设计和逻辑设计，操作系统的条带宽度、条带深度、I/O 读写方式等。
- 数据库的技术架构如何影响总体性能。

应用程序开发者应该：

- 仔细设计应用程序，使用数据库服务器提供的并行和排序工具，而不是尝试实施应用程序中的类似工具。
- 将锁的作用域和持续时间保持在最小值，避免对数据库资源的争用。
- 在应用程序中包含例程，它们可以在应用程序运行时被临时启用，以允许数据库服务器管理员监视响应时间和事务吞吐量。

- 在应用程序的设计和编码阶段，应用设计人员应考虑哪些 DB2 特性对系统有好处，并使用这些特性。还应该在该阶段对 SQL 语句和存储过程进行调整优化。同时做好数据库物理设计和逻辑设计，用最合适的数据库技术来解决应用中的业务需求(例如表分区、物化查询表、表压缩等技术)。

数据库用户应该：

- 注意性能并及时向数据库服务器管理员报告问题。
- 在调度大的决策支持查询和请求时保持谦让，尽可能使用少量的资源来完成工作。

有了良好的系统设计，你就可以在应用系统的生命周期中最小化性能调整的代价。最有效的调整时间是在设计阶段，在设计期间的调整能以最低的代价换来最大的收益。当然，即使在设计很好的系统中，也可能有性能降低。但这些性能降低应该是可控的和可以预见的。

但是很遗憾的是，国内的大部分应用系统往往是上线运行一段时间后，才发现很多性能问题都是由于最初设计阶段没有好好地进行调整导致的。所以，性能调整其实跨越整个应用系统的开发生命周期。

1.5　性能调整准则

以下准则可帮助您制订调整性能的总体方案。

1. 记住递减返回定律：最大的性能收益通常来自于最初的努力，以后的更改通常只能产生越来越小的效益，并且需要更多投入。

2. 不要只为调整而调整：进行调整是为了解决性能瓶颈。如果调整的资源不是造成性能问题的主要原因，那么这种调整对性能问题几乎不产生影响，而且这种调整会干扰您的判断，为后续的调整工作带来困难。因此调整的关键在于对症下药，找到影响性能因素的关键点。

3. 考虑整个系统：永远不要片面地调整某个参数。在进行任何调整前，务必考虑此次调整将对整个系统带来的影响。要综合地考虑系统硬件、存储设计、数据库配置参数、数据库的物理设计和逻辑设计、统计信息、中间件配置参数和编程技术。

4. 一次更改一个参数：不要一次更改多个性能调整参数。即使您能肯定所有更改都有好处，也没有任何办法来评估每个更改所带来的影响。每次调整一个参数来改进一方面时，几乎总是会影响至少一个您可能没有考虑到的其他方面。因而，如果一次更改多个参数，

就不能有效地判断每个更改的利弊。通过一次更改一个参数，您就可以使用基准程序来评估是否需要进行该更改。

5. 按级别测量和重新配置：和一次只应更改一个参数的理由一样，一次也只能调整系统的一个级别。建议使用以下系统级别列表作为参考：

- 硬件
- 操作系统
- 应用服务器
- 数据库管理器
- 应用程序
- SQL 语句

6. 检查是否存在硬件和软件问题：某些性能问题可通过维修硬件或修订软件来解决。如果这样可以解决问题，就不需要花过多时间来监视和调整系统。

7. 在升级硬件前搞清楚问题：虽然增加存储器或提高处理器能力可以立即改善性能，但也应花时间了解系统的瓶颈所在。因为可能花钱增加磁盘存储器后，才发现系统没有处理能力或没有可利用它的通道。

8. 在开始调整前制定回退方案：正如前面所讲，某些调整可能产生意外的性能结果。如果此调整使性能降低，应撤销该调整，改试另一种调整。如果保存了以前的设置并可重新调用它，那么撤销不正确的调整将变得非常容易。

1.6 性能调整的方法和过程

1.6.1 性能调整的步骤

不同类型的应用程序有不同的性能要求。性能改进过程可以通过以下几个步骤来考虑。

(1) 首先执行下面的初始检查：

- 获取直接用户的使用反馈，确定性能目标和范围。
- 获取性能表现好与差时的操作系统、数据库、应用统计信息。
- 对数据库做一次全面健康检查。

(2) 根据收集到的信息，以及对应用特性的了解，构建性能概念模型，明确性能瓶颈

所在，以及导致性能降低的根本原因。

- 首先应该排除操作系统、硬件资源造成的瓶颈。
- 然后针对数据库系统性能进行分析。
- 必要时还需要检查应用日志，因为系统性能问题也有可能是由于应用非 SQL 部分造成的瓶颈。

(3) 提出一系列有针对性的优化措施，并根据它们对性能改善的重要程度排序，然后逐一加以实施。不要一次执行所有的优化措施，必须逐条尝试、逐步对比。

(4) 通过获取直接用户的反馈，验证调整是否已经产生预期的效果。否则，需要重新提炼性能概念模型，直到对应用特性的了解能更加准确。

(5) 重复上述步骤，直到性能达到目标或由于客观约束无法进一步优化。

1.6.2　性能调整的限制

调整仅能够对系统的效率进行一定程度的更改。您同时还要考虑需要投入多少时间和费用来改善系统性能，以及需要投入多少额外的时间和费用来帮助系统的用户。

例如，如果系统遇到性能瓶颈，通常可以通过调整来提高性能。如果已接近系统的性能限制，而用户数大约增加 10%，那么响应时间可能远远不止增加 10%。在这种情况下，需要确定如何调整系统来抵消降低的性能。

但是，有一个临界点，超过这个临界点再进行调整就没有用了。达到该临界点时，应考虑在环境的限制下调整您的目标和期望值，而不是一味地增加更多的 CPU、存储和内存。

1.6.3　向客户了解情况

需要对系统进行调整的首个征兆可能是用户提出的意见。如果您没有足够的时间来设定性能目标并通过一种完备的方式来监视和调整，可听听用户的意见。通常可提出以下几个简单的问题，确定从哪里开始查找问题。例如，可以询问用户：

- 应用是 OLTP、OLAP、DSS 还是批处理等？
- “响应慢”达到何种程度？是比预期的慢 10% 还是慢数十倍？
- 问题是何时发现的？它是最近出现的，还是一直都存在？
- 其他用户有相同问题吗？这些用户是一两个人还是整个一组？
- 如果是一组用户遇到相同问题，他们是否与同一个局域网连接？

- 这些问题似乎与特定事务(例如银行结息日、结算日等)或应用程序相关吗？
- 注意到问题出现时有任何规律吗？例如，问题是否是在一天的特定时间发生，如上午刚开始上班的时间，或持续的时间有多长？
- 硬件环境。
- 操作系统版本和补丁情况。
- 应用服务器(WebLogic、WebSphere)和交易中间件(TUEXDO、CICS 和 MQ 等)。
- 数据库(Oracle、DB2、Sybase 和 Informix)。
- SQL 语句。
- 应用开发模式(C/C++、Java、嵌入 SQL 编程、CLI、J2EE、MQ 或 CICS 等)。

1.6.4 性能调整流程图

以我们本章开始时所举的案例为例，在这个应用系统中，存储在最底层，然后依次是操作系统、数据库、应用服务器和应用程序。所以我们在调整的时候应该遵循由下向上、由粗到细的原则，具体调整流程如下：

(1) 检查手机银行系统中所有硬件系统，特别是 SAN 网络中的硬件，确认硬件是否存在问题。(存储 I/O)

(2) 检查 SAN 交换机的数据流量，观察是否存在通道流量不对称、数据包丢失或数据传输过程中校验错的问题。(存储 I/O)

(3) 分析 ESS 上的数据分布，利用相关存储监测软件，观察是否存在 FC 通道、cluster、HBA 卡负载不平均的现象。(存储 I/O)

(4) 检查 CPU、内存、I/O 和网络，确定是否存在资源性能瓶颈。(操作系统)

(5) 监控数据库运行，检查 DB2 数据库的参数设置是否合理。(数据库)

(6) 监控应用服务器，确认相关配置参数是否合理。(中间件)

(7) 确定最影响性能的应用程序，协助软件开发商优化应用程序。(应用程序)

流程的前 3 个步骤是存储层面的。因为存储厂商不同并且各自需要专门的存储监控软件，而这超出了我们的讨论范围，所以下面我从操作系统入手，简单讲解如何定位性能瓶颈并做出优化。

当应用系统出现性能问题时，首先要从操作系统入手，确定是硬件故障还是系统出现资源短缺。操作系统故障定位流程图如图 1-4 所示。

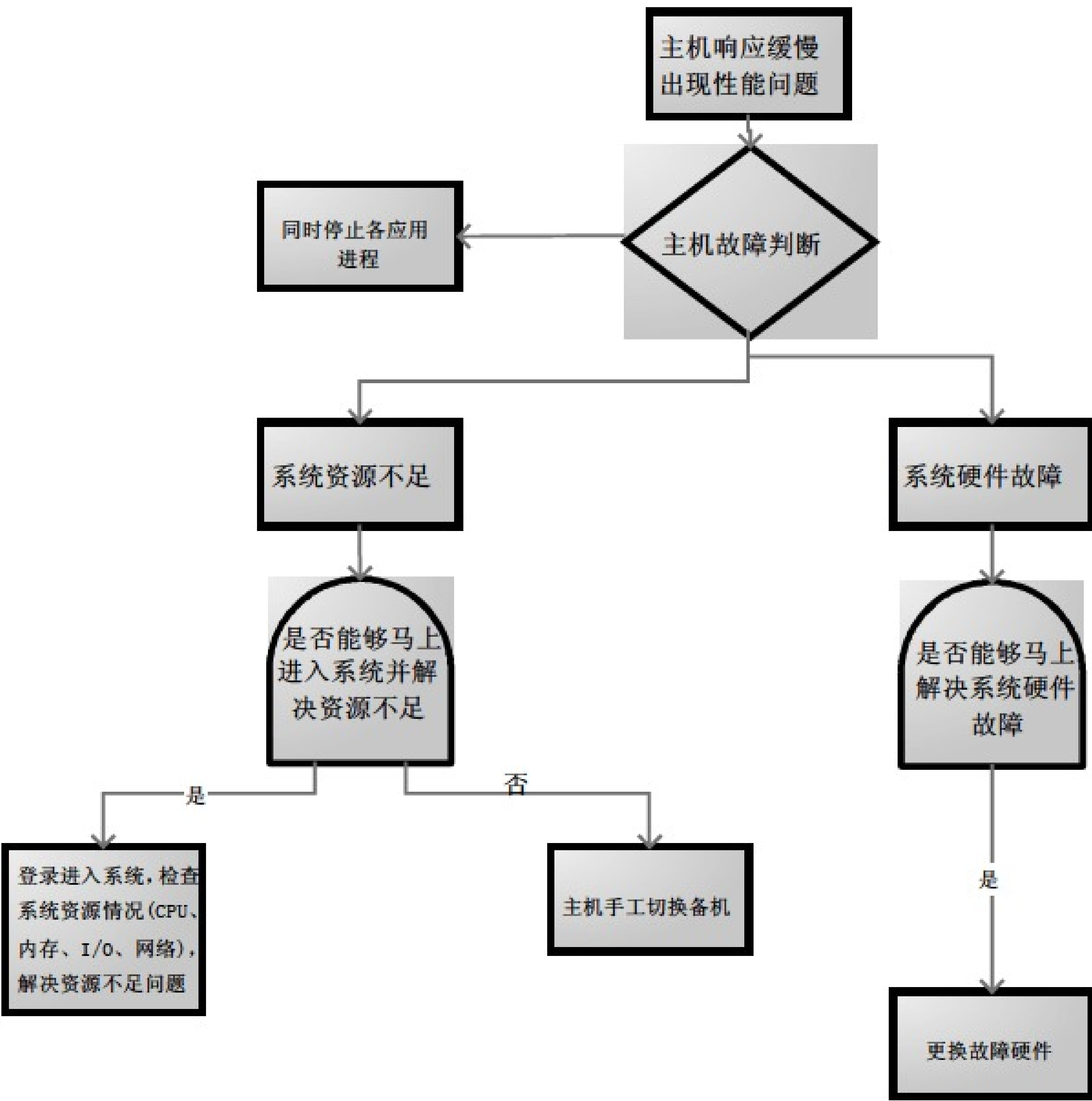


图 1-4　操作系统故障定位流程图

当系统不存在硬件故障而存在资源瓶颈时，可以参考图 1-5(AIX)和图 1-6(HP-UX)来定位资源瓶颈所在。

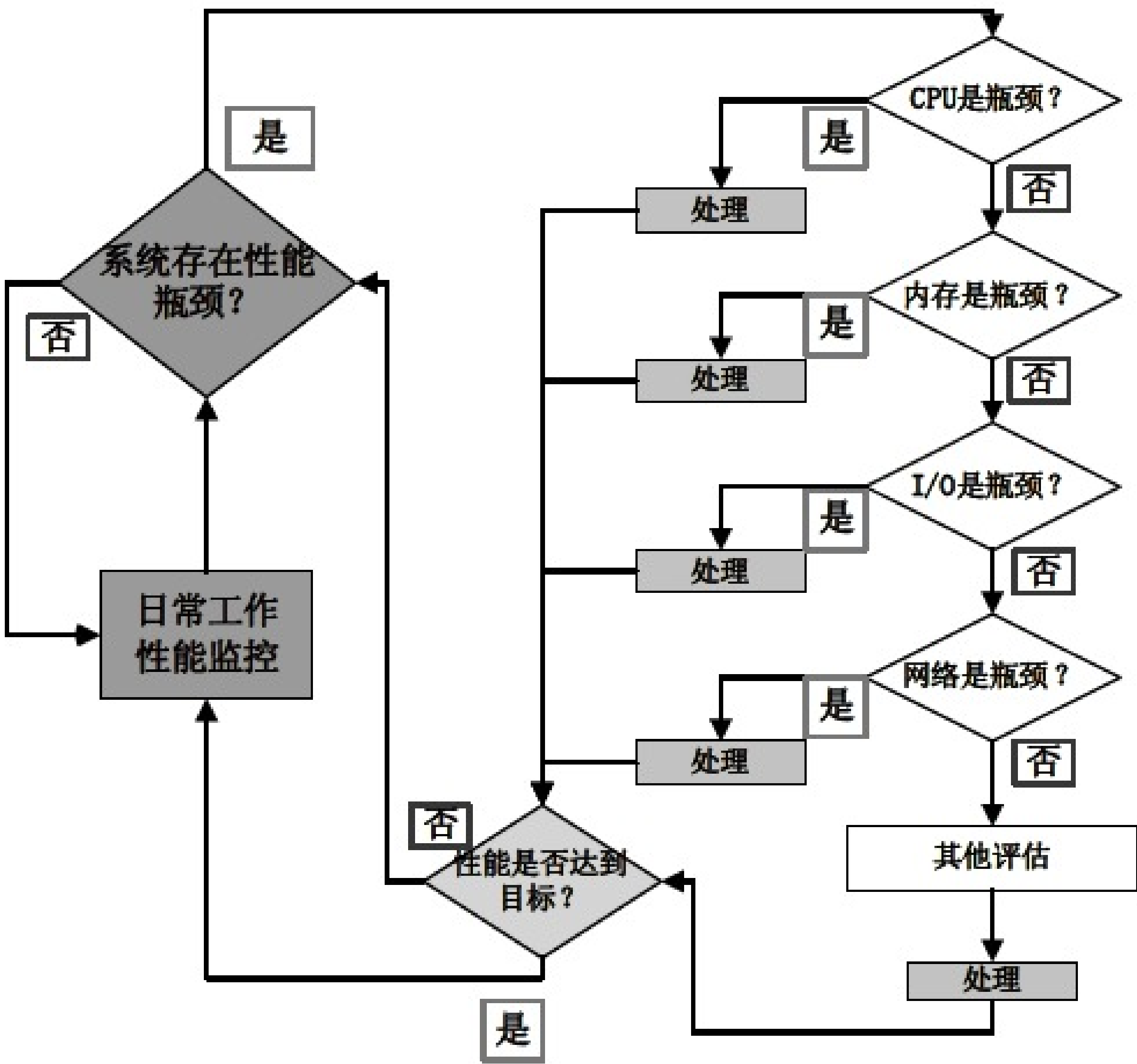


图 1-5　操作系统(AIX)性能调整流程图

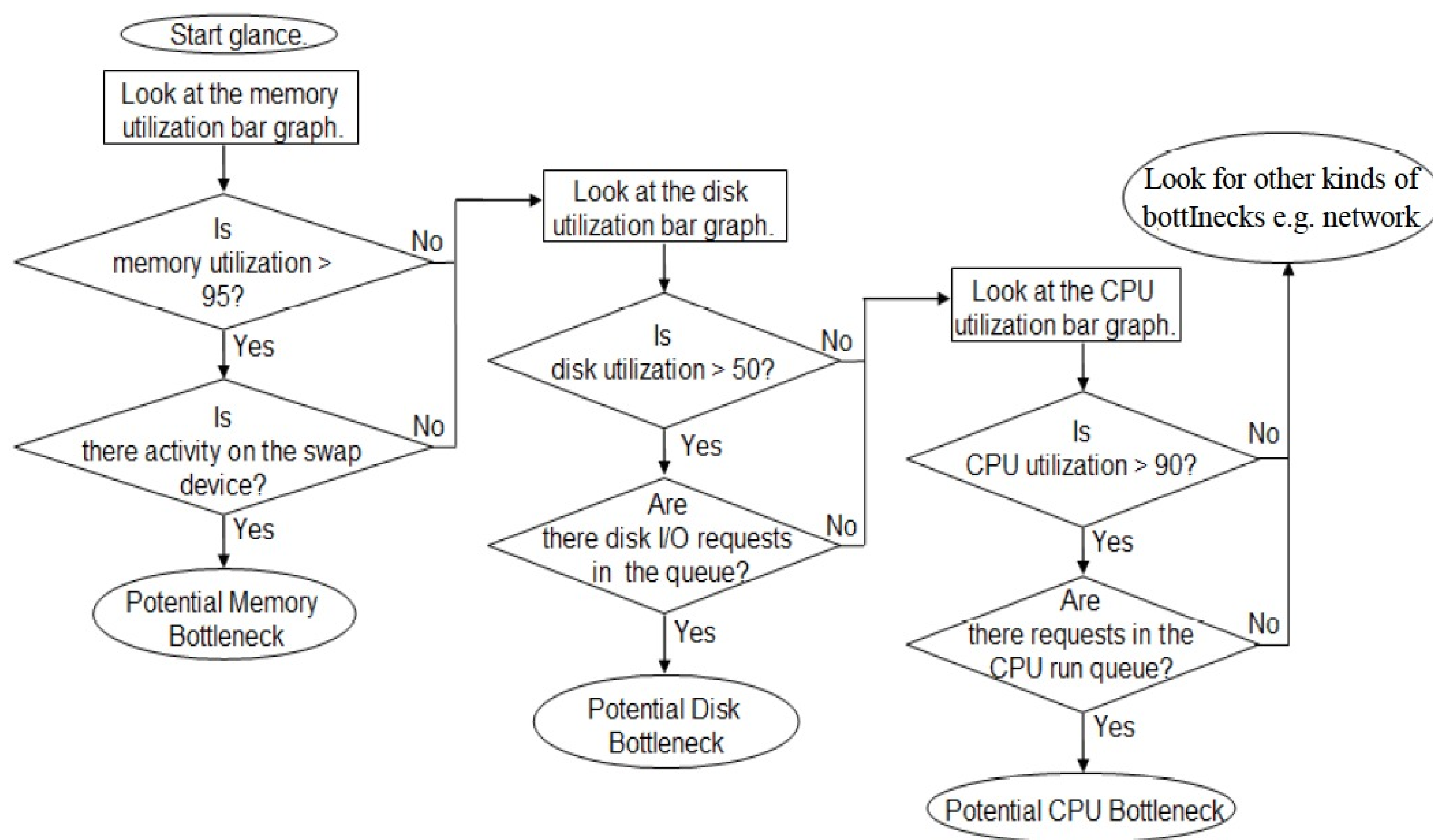


图 1-6　操作系统(HP-UX)性能调整流程图

服务器中主要有 4 类硬件资源：CPU、内存、存储、网络。检查 AIX 系统性能状态的主要目标是发现系统配置的硬件资源是否满足当前应用负载的需要，是否存在某一类型资源紧张从而成为整个系统稳定高效运行的瓶颈。定位出系统瓶颈之后，可以有针对性的做些调整，一些系统内核参数可以根据应用特点做出调整来解决部分性能问题，有些系统性能问题则需要通过升级硬件或调整应用来解决。

如果从操作系统层面判断出系统存在瓶颈并且是由数据库引起的，那么可以按图 1-7 所示的流程图来解决。数据库的诊断方法也要遵循从下到上的原则，先确定瓶颈点在什么地方，是 CPU、磁盘、网络还是数据库内部的问题？然后再结合数据库的监控命令来进一步定位。如果是 CPU 出现问题，那么就要通过系统监控确认是 USER 高还是 KERNEL 高。如果是 USER 高，那么需要用数据库命令来排查原因；如果是由于并发活动太多造成的，分析造成并发活动太多的原因是不是由于某些 SQL 的执行计划不合理造成的，进而对 SQL 进行调整。不管怎样，调整通常是一系列开销。一旦确定瓶颈，就可能要牺牲一些其他方面的指标来得到所要的结果。例如，如果 I/O 有问题，您可能需要购买更多内存或磁盘。

如果不可能买，那么可能要限制系统的并发性以获取所需的性能。然而，如果您已经明确地定义了性能目标，那么用什么开销来交换高性能就变得很容易，因为您已经确定了哪些方面是最重要的，如果您的目标为高性能，那么就以空间换取时间，牺牲一些空间资源来换取高性能。

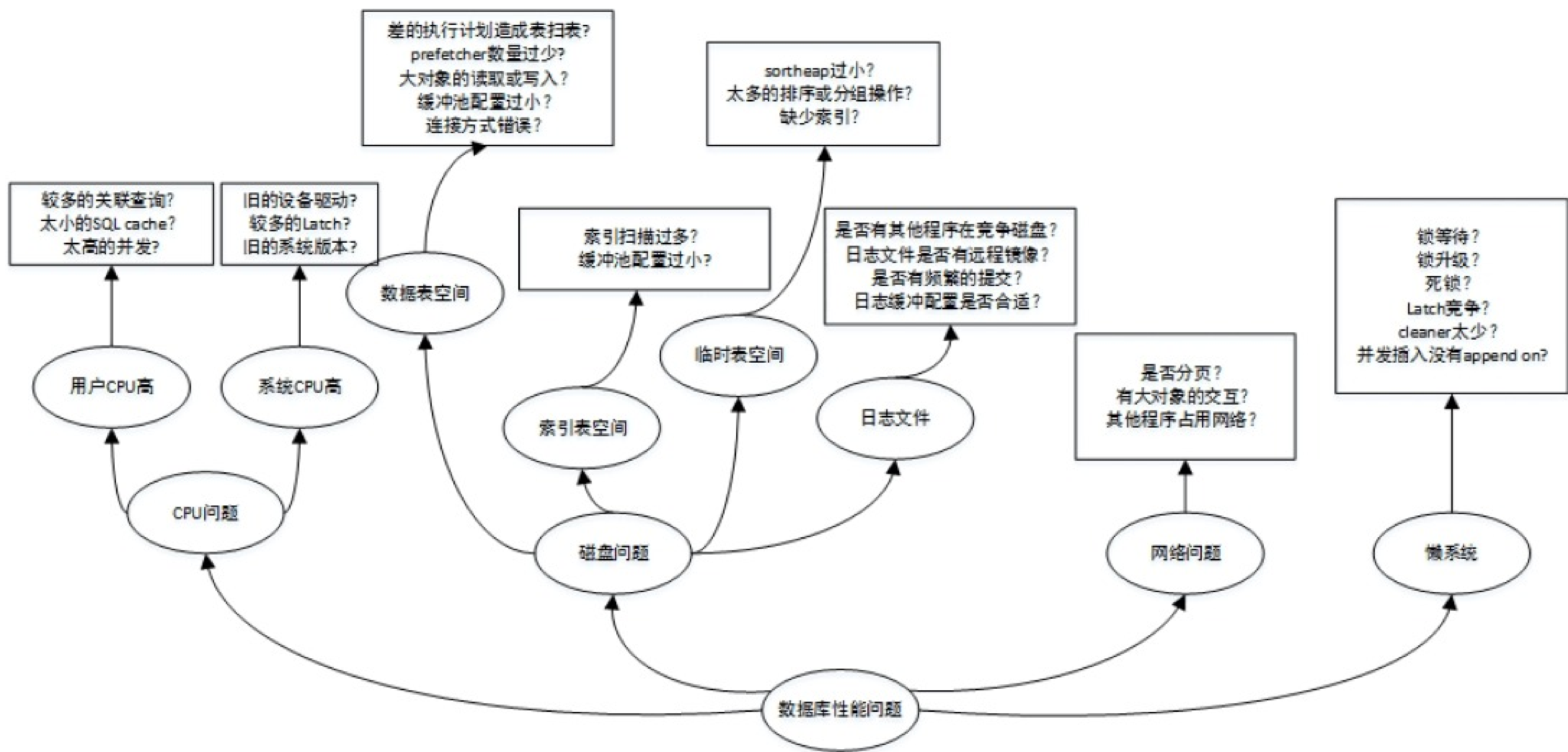


图 1-7　数据库性能调优流程图

随着应用越来越庞大，硬件性能的提高，全面的调整逐渐变成代价高昂的行为。在这种情况下，要取得最大的效率/投入比，较好的办法是调整应用的关键部分，使其达到比较高的性能，这样从总体上来说，整个系统的性能也是比较高的。这也就是有名的 20/80 原则，调整应用的 20%(关键部分)，能解决 80%的问题。

性能调整实际上就是对资源的调整，通过资源之间的交易达到您的性能目标。例如，要改进数据库的 I/O 性能，可以增加数据库缓冲池的大小。但更大的缓冲池也需要更多的内存，这就可能影响操作系统行性能和刷新内存的时间。

1.7　性能模型

如果将系统比喻成一座功能齐全的大城市，那么数据库就是这座大城市里的基础建筑，比如城市的街区、道路、桥梁、房屋等，数据库架构师就是这座大城市的规划师，负责设计出最适宜人们居住的环境，交通作为城市的传输通道，直接影响着人们的出行效率

和生活质量。差的数据库架构会让数据库在大负载、多并发的情况下变得运行缓慢，就像糟糕的交通规划会让这座城市变得拥堵不堪一样。因此，如果在出现数据库性能的时候去考虑改变数据库架构设计，无异于为城市重新规划道路结构，这是绝对无法接受的。

在之前的篇幅里我们提到了性能的方方面面，这里我们可以对前面的知识作下归纳，抽象出数据库架构中的数据库性能模型。首先数据库架构的目的是让数据库运行够快、更稳定，同时有很强的线性扩展能力，可以快速处理更多的用户请求，因此任何形式的性能问题都可以归结为如何在最低资源利用率的情况下提高数据库吞吐量和降低响应时间问题。那么影响数据库性能的 4 个因素为：CPU、磁盘、网络和内存。根据木桶原理，只有不断对这 4 个因素进行优化配置，消除短板，才能最终达到增大吞吐量、降低响应时间的目的。

数据库性能模型为数据库架构师提供了一套调优数据库性能的方法论，从宏观上来看，我们将其抽象为输入、处理和输出三个部分，如图 1-8 所示。

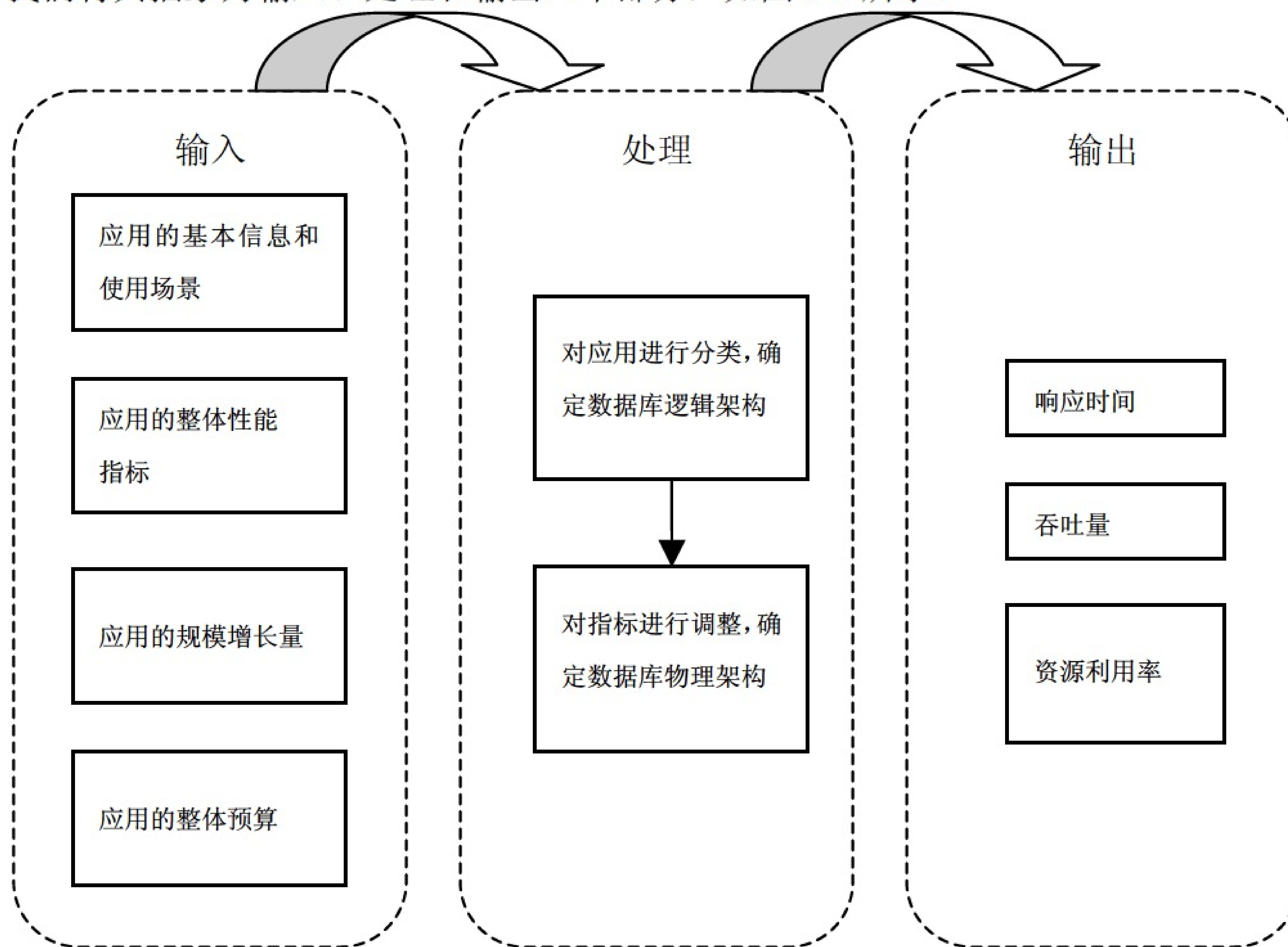


图 1-8　数据库性能模型

1.7.1　输入

输入部分里描述了数据库架构师在设计高性能的数据库之前一定要搞清楚的一件事情，即你的目标是什么？这里有句话与大家分享——“性能最优的系统不一定是最好的系统”。作为架构师，不要为了调优而调优，只要系统能达到应用的要求，就够了。希望表 1-1 中的 4 项输入可以帮助大家梳理调优目标。

表 1-1　性能调优的输入项

输　　入	描　　述
应用的基本信息和使用场景	根据应用的基本信息和使用场景，确定应用在数据库里的类型，是偏事务处理还是偏数据分析，是 CPU 密集型还是 IO 密集型或混合型
应用的整体性能指标	根据应用的整体性能指标，确定最终数据库的各项性能指标，比如： ● 缓冲池命中率 ● 各种缓存命中率 ● 读写 I/O 响应时间 ● 日志读写速度 ● CPU 使用率 ● 并发线程数 ● 网络延迟时间 ● IOPS 的值
应用的规模增长量	根据应用的增长规模，确定数据库如何适应未来的应用增长，比如： ● 数据量的增长 ● 用户数的增长 ● 业务的增加
应用的整体预算	了解应用的整体预算，有助于 DBA 知道影响数据库性能的硬件瓶颈，在软件已无太大调整空间的情况下，可以通过争取升级硬件来达到数据库性能的提升

1.7.2　处理

对于数据库架构师而言，处理可以按照两大顺序执行。首先根据应用场景和增长规模，确定数据库的逻辑架构；然后根据各种指标和预算，确定数据库的物理架构。逻辑架构是用抽象的方法描述数据库的架构，该方法脱离具体的硬件和软件，脱离具体的技术和协议，具有很强的重用性；物理架构则更多是从物理和实现上着手，根据不同的硬件、软件、技术和协议来调整数据库的参数配置，是逻辑架构的具体实现。可以说“逻辑架构决定物理架构，物理架构支撑逻辑架构”。

下面，我们先看一下数据库的逻辑架构包括哪些东西。

也许你的系统很庞杂，有很多的功能，要实现很多的业务，但是这些业务经过界面、中间件、代码逻辑处理后，到达数据库的就只会是基本的增、删、改、查操作，因此判断业务属于哪种类型很简单，大并发、小数据处理的是 OLTP 系统，小并发、大数据处理的是 OLAP 系统，两者皆有的是混合系统，不同的系统适用不同的数据库架构。这里介绍几种常见的用于处理 OLTP 和 OLAP 的数据库架构。

对称多处理器(SMP)系统由同一台机器中几个能力相等的处理器组成，像磁盘空间和内存这类资源是共享的，利用可用的多个处理器，可以更快地完成不同的数据库操作，如图 1-9 所示。该数据库系统还可将单个查询的工作分布在可用的处理器上以提高处理速度。其他数据库操作，例如装入数据、备份和复原表空间以及对现有数据创建索引，都可以利用多个处理器。

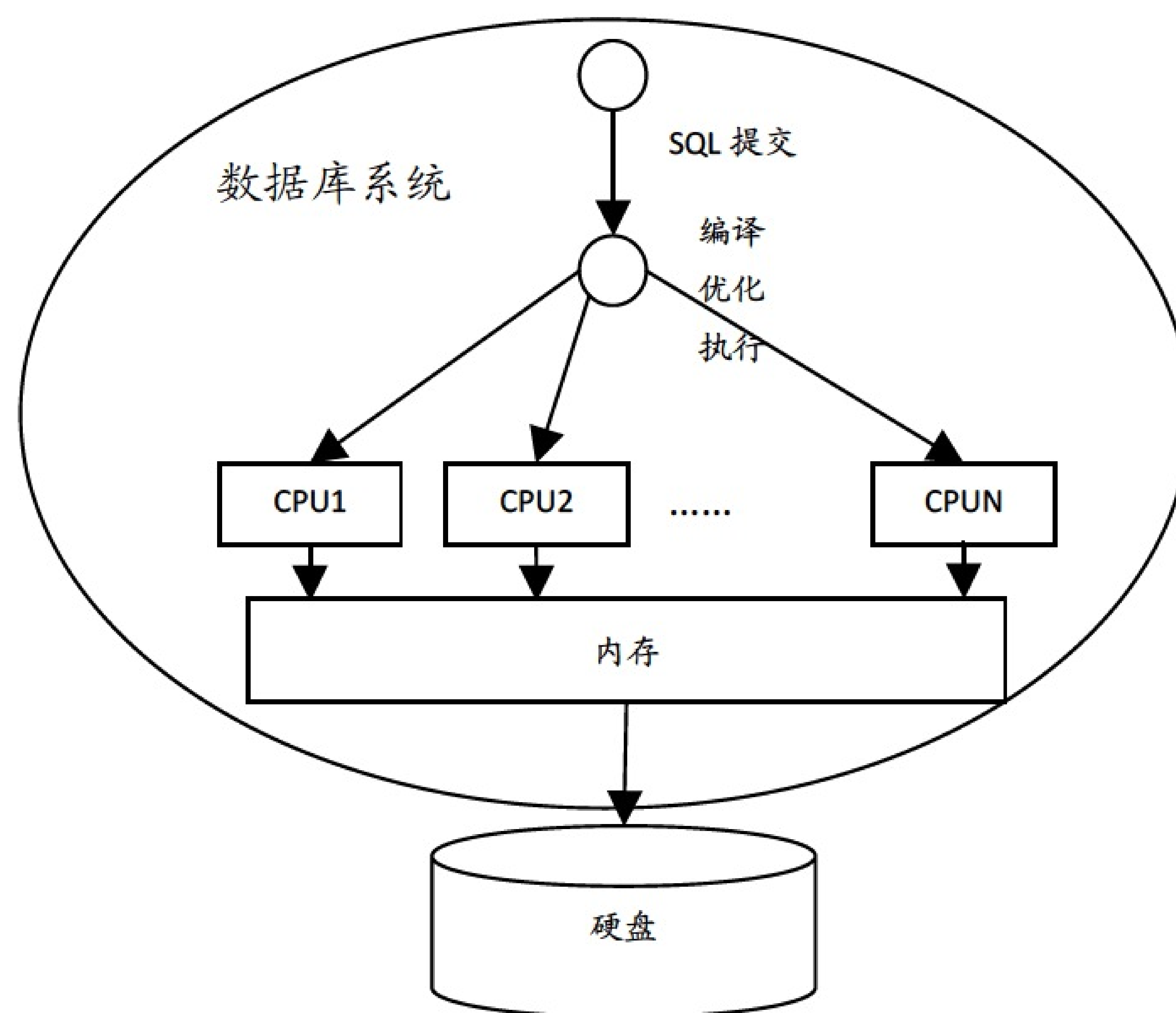


图 1-9　对称多处理器(SMP)系统

用户请求通常是以线程形式存在，在 SMP 环境中，用户的请求通过系统总线被平均分配到不同的 CPU 上进行处理，这些不同的 CPU 共享同一块内存空间和磁盘，线程之间通过信号量进行通信，因此这种结构的线程间通信的成本非常小，同时用户可以通过增加 CPU、内存或硬盘来提高整体的机器性能，非常适合处理 OLTP 类型的应用。

Sharing Nothing 架构(多分区数据库)指的是将一个数据库分成多个逻辑分区，每个逻

辑分区可以运行在不同的机器上，甚至运行在同一台机器的不同处理器上，每个逻辑分区拥有独立的处理器、内存空间、磁盘空间和日志空间，互相之间通过网络进行通信，如图 1-10 所示。用户数据通过一定的规则，比如哈希或轮询等，被分别存储到不同的数据库分区的磁盘中。不同的数据库分区之间不共享任何资源。

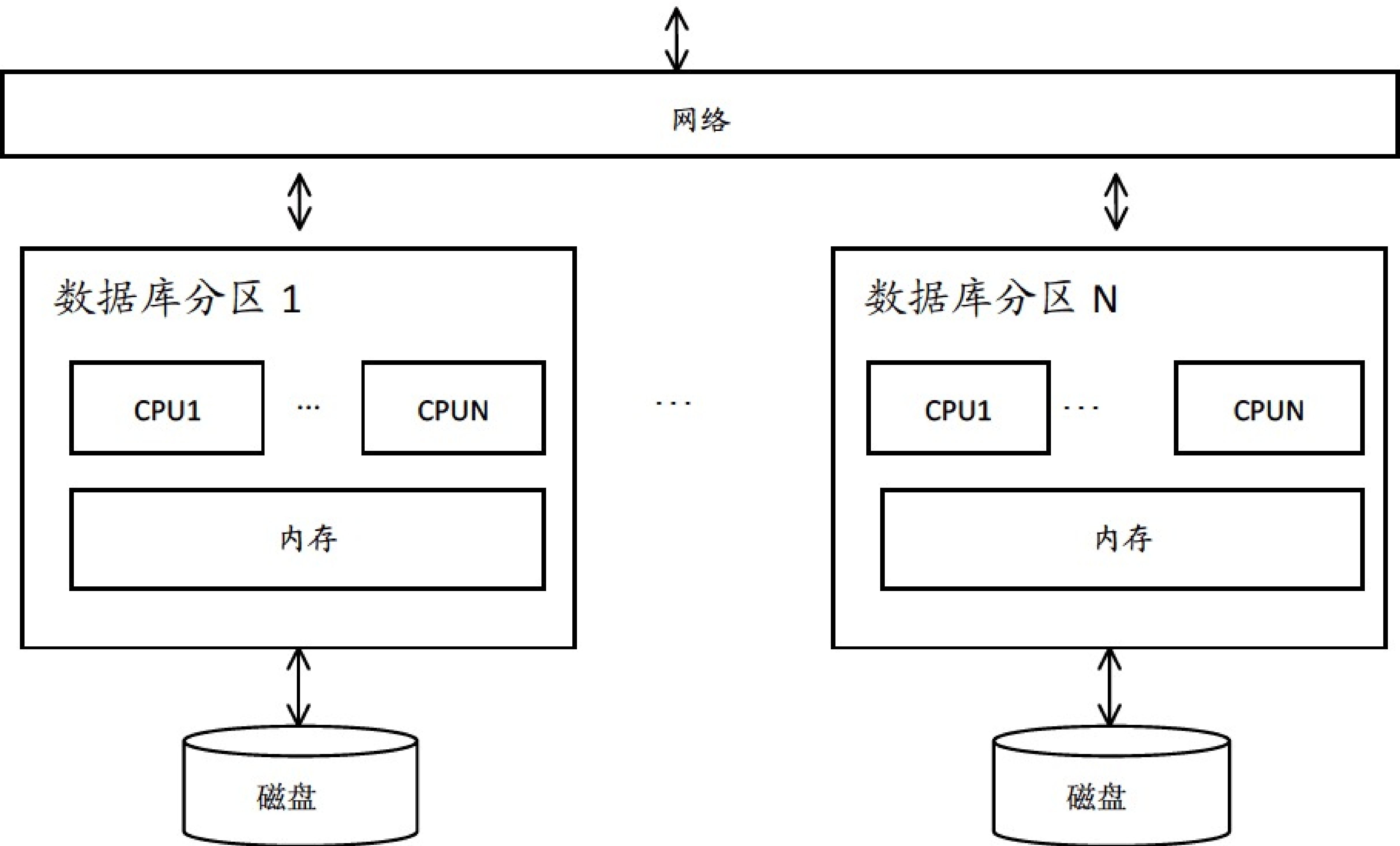


图 1-10　Sharing Nothing 架构(多分区数据库)

在这种架构下，一个用户请求被分解成多个线程，分派到不同的数据库分区并行执行，这些线程执行结束后，通过网络将结果汇总到一起，经过处理后返回给用户。用户可以根据 CPU 处理能力的不同，为数据库分区分配不同的处理器个数，做到每个 CPU 都能充分运行。因此这种架构决定了在处理大数据量复杂查询的场景下具有非常优秀的性能，相比传统的 SMP 数据库架构，在 OLAP 的用户场景中，拥有不可比拟的优势。但是由于通信网络的高延时和复杂的预处理，决定了在处理小查询、大并发、增删改操作时，时间会有明显增长，因此并不适合 OLTP 的用户场景。

然而随着业务的发展，用户数越来越多，那么能否将对称多处理器(SMP)系统进行扩展，支持多台不同的物理机器呢？答案是肯定的。Share Disk(共享磁盘)技术就是对这种扩展架构的实现。该技术可以支持更大的在线并发处理，提供良好的扩展能力和高可用性支持，其基本结构如图 1-11 所示。

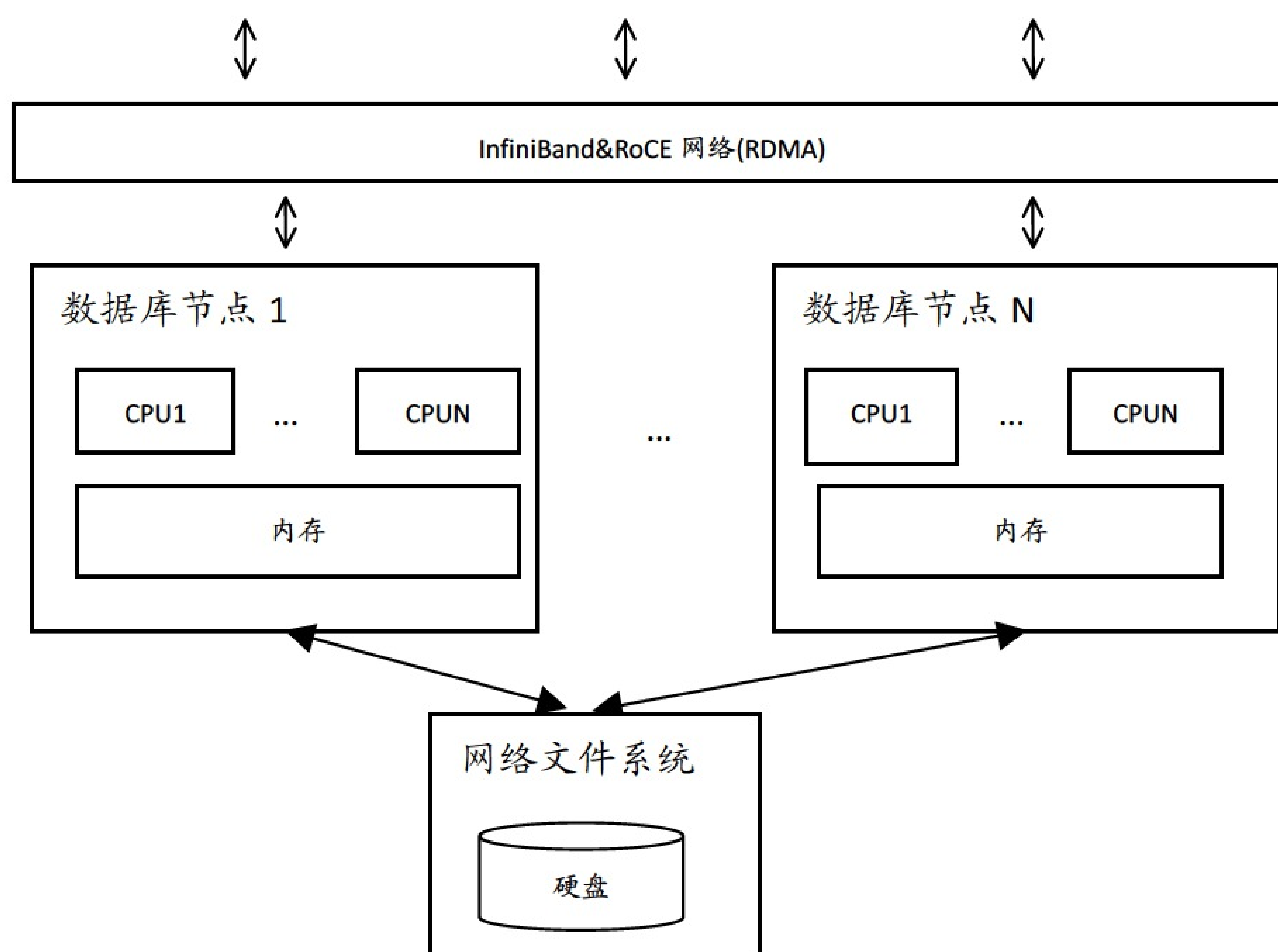


图 1-11　Share Disk(共享磁盘)技术

在该结构下，一个数据库以多个节点的形式存在，每个节点拥有不同的 CPU 和内存资源，但是共享同样的数据库磁盘，这些节点在形式上是对等的，没有主仆关系，它们可以通过网络磁盘技术同时操作数据库文件或日志文件，不同节点之间通过 InfiniBand 或 RoCE 网络的远程直接内存读写(RDMA)技术实现数据在内存中的直接读写，最大限度地避免了网络延时。由于各个节点是对等关系并且没有任何持久化的资源，因此添加节点以实现线性扩展将变得非常容易，同时任何一个节点的宕机故障也不会影响整个数据库系统的可用性。

由于篇幅有限，数据库逻辑架构我们就先谈到这里，就不再详细展开了，下面我们再来谈一谈数据库的物理架构。要想做好数据库物理架构，需要掌握下面几点：

- 掌握物理架构的方法论。
- 掌握数据库的性能指标。
- 掌握数据库的实现原理。
- 掌握各种调优方法。

就数据库物理设计的方法论而言，有下面 4 种方法。

方法 1：根据数据库逻辑设计，找到相应数据库产品的实现方法

对于同一套数据库逻辑设计，不同的数据库产品有不同的实现方法，表 1-2 列出了不同数据库产品的实现技术。

表 1-2　不同逻辑架构的实现技术

逻辑架构	实现技术
对称多处理器(SMP)	基本所有商业数据库都支持
Sharing Nothing 架构(多分区数据库)	DB2 DPF、TeraData、Greenplum、Netezza
Share Disk 架构(共享磁盘)	DB2 pureScale、Oracle RAC

方法 2：合理设计数据库对象

数据库对象是数据库的逻辑概念，其设计的好坏除了决定系统功能外，有时还会决定系统的性能，其设计主要包括模型的设计、表类型的选择、索引的设计、分区的设计、表空间的设计等。

在模型的设计上，我们通常采用第三范式来进行数据库对象的模型设计，但是有时为了性能考虑，我们通常会做一些退化，比如设计一些冗余字段或者将多表合并来加快系统的查询性能。例如对于有大量数据的销售流水表，完全按照第三范式来做的话会有一个外键关联客户信息，一个外键关联产品信息。但是如果系统在运行一段时间后，这三张表的数据变得非常多，那么连接查询的性能将会变得很差，我们通常的做法是再设计一张横向表，将客户信息和产品信息全都放到销售流水表里，这样就避免了关联查询。

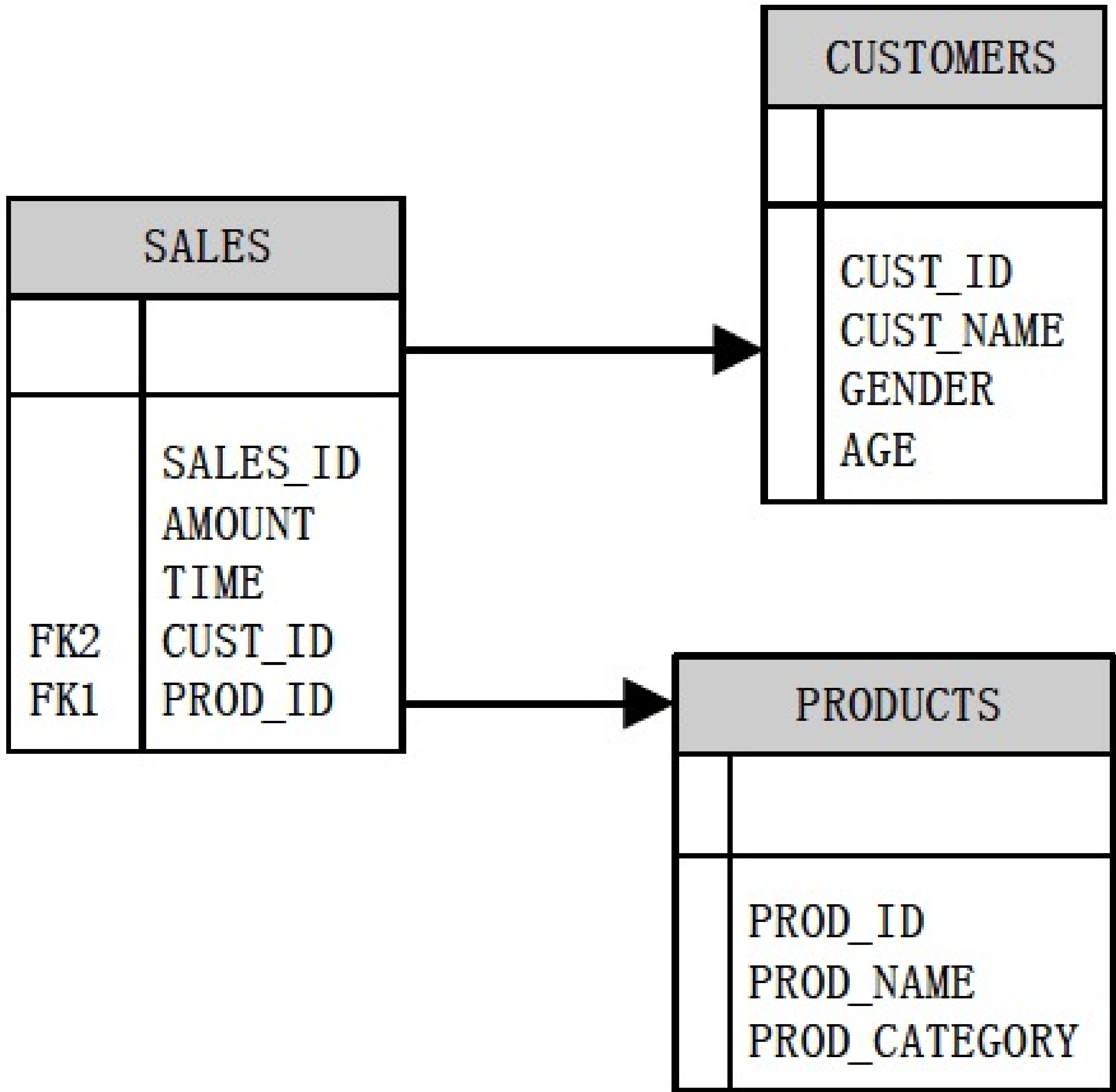


图 1-12　第三范式

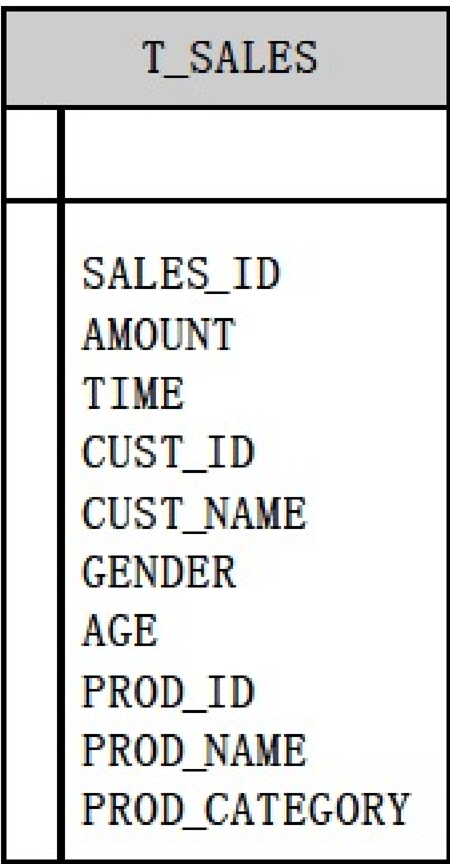


图 1-13　T_SALES 横向表

表通常分为很多种，除了基本表外，还有临时表、多维表、分区表、范围集群表、物化视图表等。不同的表有不同的用途，如果错误使用的话，也会对性能有比较大的影响。

比如在不需要物化视图的地方使用物化视图表，就会对源表的增删改性能造成影响。

合理使用正确的索引是提高系统执行效率的关键因素，对索引的使用需要注意以下一些问题：

1) 注意 LIKE 运算符。
2) 注意 NULL 值。
3) 优化复合索引。
4) 注意索引的相关参数。
5) 注意索引与谓词的关系。
6) 根据查询使用的列建立索引。
7) 根据条件语句中谓词的选择度创建索引。
8) 避免在建有索引的列上使用函数。
9) 在那些需要被排序的列上创建索引。
10) 合理使用 include 关键词创建索引。
11) 指定索引的排序属性。

数据库的页大小也是在数据库设计的时候就应该确定的，否则一旦实施就很难更改。对于执行随机行读写操作的 OLTP 应用程序，通常最好使用较小的页大小，这样不需要的行浪费的缓冲池空间就会较少。对于一次访问大量连续行的 OLAP 应用程序，页大小大一些会比较好，这样就能减少读取特定数目的行所需的 I/O 请求数，更大的页大小也可以允许您减少索引中的级别数。总之，在满足以上条件的情况下，交易系统使用较小的页更适合，仓储系统使用较大的页更适合。

对于表空间的选择，通常的数据库都会包含系统管理的表空间和数据库管理的表空间两种类型。系统管理的表空间由操作系统的文件系统管理器分配和管理空间，管理灵活但是性能很差。数据库管理的表空间由数据库管理程序控制存储空间，表空间容器可使用文件系统或裸设备，虽然管理复杂，但是在性能上会有很大的提升。

方法 3：合理设计存储

磁盘作为数据库的最底层，从发明至今，依然没有摆脱机械设备固有的速度限制，面对速度和容量突飞猛进的 CPU 和内存，存储在很多情况下依然是系统的最终瓶颈，因此作为数据库架构师，必须很好地规划存储。一般存储规划应该从下面几个方面进行：

(1) 磁盘 I/O 的吞吐量(每秒 I/O 数 IOPS 和每秒传输的数据量 MPS)。

对于磁盘密集型的系统，磁盘 IOPS 和 MPS 指标尤为重要，一次 I/O 指的是磁盘在没有换道的情况下产生读写的操作，通常一块 15 000 转的 SAS 盘的 IOPS 能在 150-175 之间，对于大并发的系统，我们应该充分估计系统峰值的 IOPS 数和 MPS 数，采用合适的 RAID 技术和磁盘数量，以防止出现 CPU IO Wait 高的情况。对于平均负载来说，通常建

议 10 到 20 块盘对应一个 CPU 的 Core。

(2) 存储的容量规划。

SATA、SAS、光纤盘，不同的盘具有不同的容量、速度和价格，如何平衡容量、速度和价格之间的关系也是前期数据库物理设计应该注意的问题。

(3) 独立磁盘记录日志。

数据库为了保证数据的一致性，在进行事务提交的时候，必须要等到数据库日志写入磁盘才算提交成功。如果日志写入缓慢的话，会造成数据库的所有事务提交缓慢，因此数据库日志最好采用独立的磁盘来单独存放，以免产生数据库整体的性能瓶颈。

(4) 条带化。

为了能够协调多块磁盘同时响应用户请求，数据库架构师应该考虑如何在多块磁盘间做条带化处理，通常的存储都已经支持 RAID0~5 和一些互相结合的条带化技术。条带的深度和宽度也应该和数据库的扩展块大小进行合理匹配。

方法 4：优化数据库参数，减少资源竞争

最后一点是优化配置数据库参数，包括各种缓存池的大小、内存区的配置、刷新脏页的策略、锁的策略等。虽然各个数据库都不相同，但是所有的出发点都是为了通过数据库参数，降低物理读的次数和发生资源竞争的可能性。这里就不再详细介绍了。

1.7.3　输出

通过上面的处理，您的数据库应该有了很大的提高，那么赶紧将调优后的数据库指标记录下来，发给项目组做最终的性能评审吧。这里总结了一些应该关注的性能指标，请大家按照需要补充。

1) 执行次数最多的 SQL 的平均执行时间。
2) 耗时最长的 SQL 的平均执行时间。
3) 数据库的平均 CPU 利用率。
4) CPU 的执行时间、IO 等待时间和锁等待时间。
5) 平均 I/O 响应时间。
6) 支持的峰值 IOPS 数和 MPS 数。
7) 物理读和逻辑读的百分比。
8) 有效索引读的百分比。
9) 有效行读的百分比。

1.8 本章小结

如果想让您的系统保持良好的性能，根据作者的调优经验，感觉除了 DB2 自身的 bug、补丁和产品缺陷(这需要 IBM 原厂商提供)外，您需要做以下几个方面的工作：

1. 有足够的物理资源并且能合理地使用。如果系统配置充足的 CPU、内存、高速硬盘，并且有足够的网络带宽，那么这个系统就可以很好地保证我们应用的系统。同时，在物理资源一定的情况下，如何合理使用它们也是性能调整的一个方面。例如，使用异步 I/O、使用裸设备和使用并发 I/O(CIO)等。关于这部分的详细内容，参见本书第 2 章。

2. 良好的存储 I/O 设计。必须有足够的磁盘设备来确保充分的 I/O 并行性，以支持大容量的并发事务。对于中等工作负载而言，每个 CPU 至少应当有 5 到 10 个磁盘，对于高 I/O OLTP 工作负载而言，至少要有 20 个磁盘。操作系统(包括交换空间)、DB2 日志和 DB2 表空间应当拥有各自的专用磁盘。应当有多个磁盘用于 DB2 日志、表和索引。

估计良好性能所需的 I/O 处理能力的正确方式，实际上是制作事务原型并找出每个事务需要多少 I/O，以及每秒需要处理多少事务。然后找出磁盘控制器和磁盘子系统的 I/O 速率，以帮助确定需要多少控制器和磁盘。关于这部分的详细内容，参见本书第 2 章。

3. 合理的数据库配置参数。数据库配置参数影响数据库资源的分配，合理的数据库配置参数可以充分发挥资源优势，使数据库在最合理、最优的情况下运行。关于这部分内容，参见本书第 4 章。

4. 为表创建最合理的索引。确保查询中进行连接操作的列都有索引。如果为 ORDER BY 和 GROUP BY 涉及的列建立了索引，那么就可以提高性能。也可以将经常被访问的数据作为 INCLUDE 子句中的列包含在索引中，根据所使用的表和 SQL 语句，使用索引顾问程序 Index Advisor(也称索引向导，Index Wizard)创建一组合适的索引。确保使用合适的索引，从而在选择和连接(join)表时，将必须在内部访存的行数减到最少。关于这部分内容，参见本书第 6 章。

5. 确保应用程序持有锁的时间尽可能短。当用户操作涉及多个交互时，每个交互应当提交自己的事务并在将活动返回给用户之前释放所有锁。通过尽可能晚地启动事务的第一条 SQL 语句(可以启动事务)并使事务的更新(插入、更新和删除，这些操作要用到互斥锁)尽可能接近提交阶段，使得事务的持续时间尽可能地短。减少锁等待、锁升级和死锁，可以提高并发性能。关于这部分内容，参见本书第 5 章。

6. 高效地使用 SQL 语句。通常，如果一条 SQL 语句能完成任务，就不必使用多条 SQL 语句。当通过在查询中设置更多谓词来提供更详细搜索条件时，优化器就有机会做出更好的选择。您还应该使查询具有可选择性，这样数据库就不会返回您不需要的行和列。

例如，使用 SQL 来过滤您想要的行；不用返回所有行，然后要求应用程序执行过滤操作。关于这部分内容，参见本书第 9 章。

7. 分析 SQL 执行计划。分析一条 SQL 语句的执行计划可以帮我们找出该 SQL 语句的运行瓶颈并做出调整，可以使用 db2exfmt 和 db2expln 来分析每一条 SQL 语句。关于本部分内容，请参见第 9 章。

8. 数据库的物理和逻辑设计。利用最合理的数据库技术来满足业务逻辑设计，例如表分区、数据库分区、MDC(多维群集)、物化查询表、表压缩和 XML 等先进的数据库技术。在设计业务系统时，一定要用最合理的数据库技术来实现我们的业务逻辑，良好而合理的设计往往是整个系统高效运行的必要条件。

9. 统计信息更新、碎片整理。必须定期更新统计信息，以使优化器时刻为 SQL 制定最合理的执行计划。同时，也应该定期对频繁插入、删除的表做碎片整理以提高性能。关于这部分内容，参见本书第 8 章。

10. 了解 DB2 优化器的工作原理。优化器是 DB2 数据库的心脏和灵魂，了解它的工作机制和组件可以使我们更好地使用它，从而为我们制订更合理的访问方案。关于这部分内容，参见本书第 7 章。

第 2 章

操作系统及存储的性能调优

在第 1 章，我们举了一个某银行手机银行系统的例子，该系统比较复杂，那么我们来看看该系统的整体 I/O 流动图，看一下业务访问的整个过程，如图 2-1 所示。

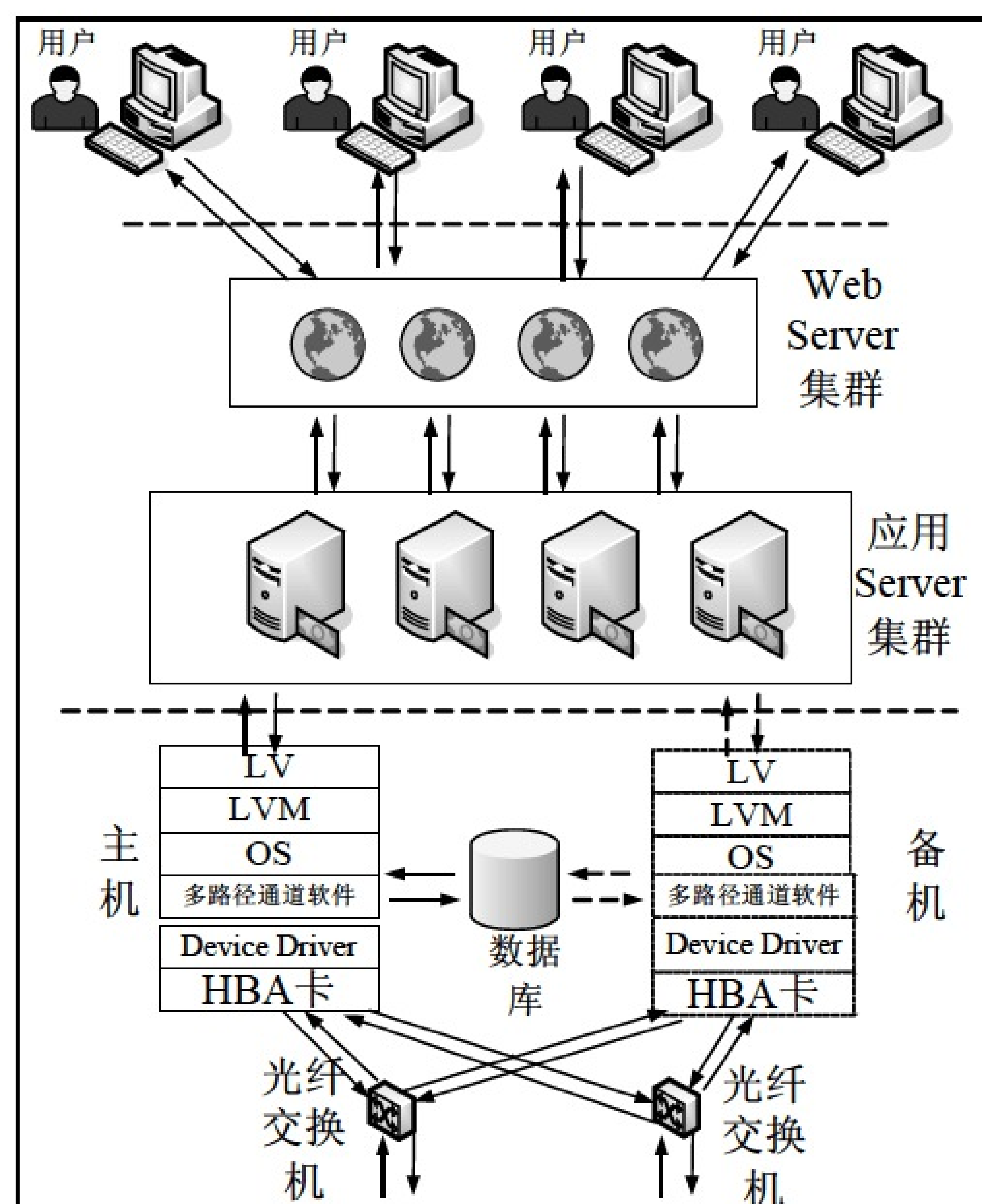


图 2-1　I/O 流动图(a)

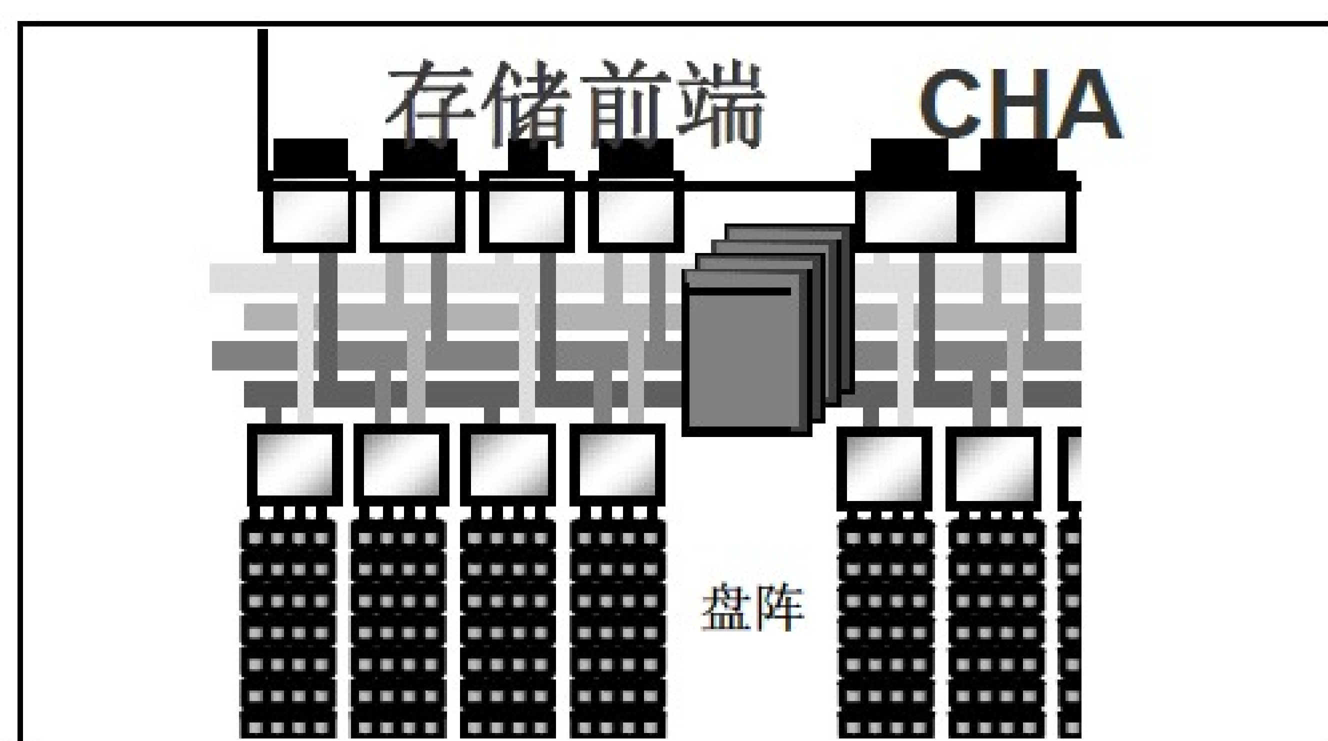


图 2-1　I/O 流动图(b)

图 2-1 展示了该行手机银行系统的数据处理层次图。在这个数据处理层次图中，我们可以首先简单地将其划分为三个较大的层次：用户层、应用层、数据层。

- 用户层：用户负责发出对数据的请求。
- 应用层：提供各种满足要求的应用，将用户的请求向数据层传递，同样负责将数据层获得的数据处理后发送给用户。
- 数据层：负责数据的管理以及最终的数据 I/O。

对于性能的问题，应用层和数据层都需要引起我们的关注，这两者都会对整个系统的性能产生较大的影响。

假设最终用户在前台发出了一个查询事务，该事务其实就是一条 select 语句，那么该请求通过 http/https 发送到 Web Server 服务器，Web Server 接收到该请求后再把该请求转发给 Application Server，在这个阶段我们要确保 Web Server 方面不存在瓶颈，否则它无法及时地把最终用户请求转发，会引起事务请求阻塞；Application Server 接收到请求后，通过应用程序业务逻辑去数据库中检索记录，在该阶段，我们要确保 Application Server 不存在瓶颈，不会阻塞到数据库的请求，否则我们要进行 Application Server 调优；当这个请求到达 DB Server 后，数据库优化器会为该 SQL 语句进行语法检查、语义分析、权限检查、查询重写等，之后会制订最优的访问计划(access plan)，在该计划中决定用什么索引、用什么扫描方式、访问表的顺序和连接方式等。假设优化器为这条 SQL 语句确定的是全表扫描方式，那么它首先判断数据库 bufferpool 中是否有它请求的数据，如果有，就直接从内存读取，这叫逻辑读；否则就要从硬盘读，那么要首先判断该表在哪个表空间中，如果该表空间采用的是裸设备，那么它会去读逻辑卷(lv)，逻辑卷是在逻辑卷管理器(lvm)之上的，而 lvm 又在操作系统之上；在操作系统和存储之间往往还会有存储的多路径通道软件(例如 AIX 上的 MP I/O Driver、EMC 的 power path、日立的 HiCommand Dynamic Link Manager(HDLM)等)，可以实现动态负载均衡和流量控制；在多路径通道软件之下是存储供应商的设备驱动(Device Driver)；主机服务器通过 HBA 卡和光纤交换机连接，光纤交换机和存储的前端连接，存储的前端再通过存储的 Cache 连接后端的物理磁盘。这就是一次磁

盘 I/O 的处理过程。在上面的 I/O 流动中，每个环节都有可能出现性能问题，所以大家要能够理解这个 I/O 过程，在出现性能问题时，能快速地找出是哪个环节引起了性能瓶颈。

在本章中，我们将重点介绍数据库层以下的性能调优，即操作系统和存储的性能调优。首先给大家介绍如何在操作系统(AIX)上通过性能监控工具来定位资源的瓶颈，以及如何合理地采用操作系统的相关技术(限于篇幅主要讲解 AIX 操作系统)来提高性能；然后再介绍一下存储的相关概念、存储 I/O 的设计以及如何合理地采用存储的相关技术来提高性能。

本章主要内容包括：

- AIX 性能监控综述
- 操作系统性能优化
- 逻辑卷和 lvmo 优化
- 操作系统性能调整总结
- 存储 I/O 设计
- 存储基本概念
- 存储架构
- 良好存储规划的目标
- 良好存储规划的设计原则
- 存储相关性能调整案例
- 存储 I/O 性能调整总结

2.1　AIX 性能监控综述

AIX 操作系统中同样也有 4 类可能的资源瓶颈：CPU、内存、存储、网络。检查 AIX 系统性能状态的主要目标是发现系统配置的硬件资源是否满足当前应用负载的需要，是否存在某一类型资源紧张，从而成为整个系统稳定高效运行的瓶颈。定位出系统瓶颈之后可以有针对性地进行调优，同时可以根据应用特点对一些系统内核参数进行调整以解决部分性能问题，有些系统性能问题则需要通过升级硬件或调整应用本身才能解决。

2.1.1　监控工具

AIX 操作系统提供了丰富的命令工具来检查系统性能状态，其中最重要的命令是 topas，它提供了实时系统的主要硬件资源运行状态的监控界面。在 topas 监控界面中，通过切换不同的视图可以详细显示某一特定资源的运行状态。分析 topas 提供的实时监控数据，可以判断出当前系统上是否有性能问题，以及造成性能瓶颈的硬件资源，然后利用系统提供的其他工具有针对性地分析某类硬件资源的运行状态。

- 检查 CPU 状态的命令有 sar、vmstat 和 tprof。
- 检查内存状态的命令有 vmstat、lsps、ipcs、ps 和 svmon。
- 检查存储状态的命令有 iostat、df 和 dd。
- 检查网络状态的命令有 netstat。

2.1.2 监控系统总体运行状态

首先从 topas 命令入手来查看系统是否运行状态良好，是否存在性能问题。启动 topas 监控界面，如图 2-2 所示。topas 默认的刷新间隔为 2 秒，使用 topas -i <n>命令可以更改刷新间隔为 n 秒。

```
Topas Monitor for host:    ARAPPSQ                EVENTS/QUEUES      FILE/TTY
Wed Oct 10 10:31:31 2012   Interval:  2           Cswitch      342   Readch      266
                                                  Syscall     1507   Writech    8509
CPU   User%  Kern%  Wait%  Idle%                  Reads          0   Rawin         0
ALL    0.1    0.2    0.0   99.8                   Writes         5   Ttyout      266
                                                  Forks          0   Igets         0
Network  KBPS   I-Pack  O-Pack   KB-In  KB-Out    Execs          0   Namei         0
Total     4.0    20.9     9.0     2.2     1.8     Runqueue     0.0   Dirblk        0
                                                  Waitqueue    0.0
Disk    Busy%     KBPS     TPS KB-Read KB-Writ                       MEMORY
Total     0.0      0.0     0.0     0.0     0.0    PAGING             Real,MB   31488
                                                  Faults         0   % Comp     41
FileSystem        KBPS     TPS KB-Read KB-Writ    Steals         0   % Noncomp  29
Total              8.0     4.0   0.3     7.8      PgspIn         0   % Client   29
                                                  PgspOut        0
Name            PID  CPU%  PgSp Owner             PageIn         0   PAGING SPACE
db2sysc      450754   0.1 190.2 arapdb            PageOut        0   Size,MB   32768
dtgreet      155678   0.0   1.3 root              Sios           0   % Used        0
xmgc          45078   0.0   0.4 root                                 % Free      100
java         115054   0.0 674.4 weblogic          NFS (calls/sec)
topas        143806   0.0   1.9 root              SerV2          0   WPAR Activ    0
gil           25156   0.0   0.9 root              CliV2          0   WPAR Total    0
swapper        4384   0.0   0.4 root              SerV3          0   Press: "h"-help
swapper        4920   0.0   0.4 root              CliV3          0          "q"-quit
swapper        4652   0.0   0.4 root
db2fmp       107264   0.0  16.8 arapdb
python        62140   0.0  40.0 root
sshd         147868   0.0   1.1 root
rpc.lock      82928   0.0   1.2 root
X             50050   0.0   5.0 root
sendmail      65772   0.0   1.1 root
db2fmcd      475632   0.0   1.2 root
topas_nm     254426   0.0   3.4 root
```

图 2-2　topas 监控界面

在 topas 监控界面的左上角可以观察到操作系统的主机名、当前系统时间和监控数据刷新间隔。

然后依次观察 CPU、内存、I/O、网络的运行状态。图 2-3 中的框选部分为 CPU 的性能状态数据。

```
Topas Monitor for host:    ARAPPSQ              EVENTS/QUEUES    FILE/TTY
Wed Oct 10 10:36:09 2012   Interval:  2         Cswitch     401  Readch   953.0K
                                                Syscall    2993  Writech    8288
Kernel     0.3   |#                          |  Reads       236  Rawin         0
User       0.2   |#                          |  Writes        6  Ttyout       82
Wait       0.0   |#                          |  Forks         1  Igets         0
Idle      99.5   |############################|  Execs         2  Namei       576
                                                Runqueue    0.0  Dirblk        0
Network  KBPS    I-Pack  O-Pack   KB-In  KB-Out Waitqueue   0.0
Total    10.4      90.5    83.0     5.4     5.0                  MEMORY
```

图 2-3　CPU 性能状态数据

CPU 资源被划分成 Kernel、User、Wait、Idle 4 个部分：Kernel 和 User 部分分别表示 CPU 资源用于 Kernel 模式和用户模式的两种 CPU 运算时间，在 CPU 繁忙的情况下主要会表现为这两部分的占比较高；Wait 部分表示 CPU 在等待存储或网络资源，此部分持续较高表示系统存储或网络资源存在瓶颈；Idle 部分反映 CPU 资源空闲的比例，当 Idle 部分数值接近 0 时表示系统 CPU 很繁忙，系统 CPU 资源不足，这时可以参考 2.1.3 节内容来进一步分析 CPU 资源的使用情况。

图 2-3 右侧部分的 Runqueue 表示位于等待 CPU 资源的队列中的线程数，但这是一个全局的值，而实际在操作系统中每个 CPU 都有一个运行队列，因此在评估此值时需要除以操作系统中 CPU 的数量，而当得到的结果如果持续大于 1 时，通常表示 CPU 资源不足。Waitqueue 表示已获得 CPU 资源但需要等待 I/O 资源的线程数，当 Waitqueue 的数值持续大于 1 时，通常表示 I/O 资源不足。在一个没有性能问题的系统中，Runqueue 和 Waitqueue 的值通常为 0，它们偶尔出现不为 0 的数值并不代表系统存在性能问题。

图 2-4 中的框选部分为内存状态数据。

```
Network  KBPS   I-Pack  O-Pack   KB-In  KB-Out  Waitqueue   0.0
Total     3.7     18.9     8.9     2.1     1.6              MEMORY
                                                PAGING       Real,MB    31488
Disk     Busy%    KBPS     TPS KB-Read KB-Writ  Faults     0 % Comp        41
Total     0.0      0.0     0.0     0.0     0.0  Steals     0 % Noncomp     29
                                                PgspIn     0 % Client      29
FileSystem        KBPS     TPS KB-Read KB-Writ  PgspOut    0
Total              7.9     4.0     0.1     7.7  PageIn     0 PAGING SPACE
                                                PageOut    0 Size,MB    32768
                                                Sios       0 % Used         0
                                                             % Free       100
                                                NFS (calls/sec)
```

图 2-4　内存状态数据

当 PAGING 部分的 PgspIn 和 PgspOut 项持续不为 0 时，表示系统在不停地做 page out(换页)操作，这是系统物理内存资源不足的重要标志，这时可以参考 2.1.4 节来进一步分析物理内存资源的分配和使用情况。

在 MEMORY 部分可以看到系统配置的物理内存总数，当前系统使用物理内存用于计

算内存和文件缓存部分的比例，当系统物理内存资源紧张时，可以观察到%Comp 和%Noncomp 的数值之和接近 100%，使用公式 100% - %Comp - %Noncomp 可以获得空闲物理内存的比例，此公式的计算结果如果很低，就表示物理内存是不足的。

在 PAGING SPACE 部分可以看到系统配置的全部换页空间的容量，当前已使用的与空闲的换页空间的比例，在物理内存充裕的系统上，换页空间的使用比例应接近于 0。对于换页空间的空闲比例较小的系统，需要扩展换页空间的容量，同时也表示物理内存资源紧张，应当启动内存瓶颈的分析。

图 2-5 中的框选部分显示了存储系统状态数据。

```
Topas Monitor for host:    ARAPPSQ              EVENTS/QUEUES       FILE/TTY
Wed Oct 10 10:58:03 2012   Interval:  2         Cswitch      803    Readch     79980
                                                Syscall     2114    Writech  2289.6K
Kernel      0.4   |#                        |   Reads         35    Rawin          0
User        0.2   |#                        |   Writes       112    Ttyout       133
Wait        3.7   |##                       |   Forks          3    Igets          0
Idle       95.7   |############################ |   Execs      2    Namei         44
                                                Runqueue     0.0    Dirblk         0
Network  KBPS   I-Pack  O-Pack   KB-In  KB-Out  Waitqueue    0.0
Total    14.0    117.1   114.1     7.2     6.8                      MEMORY
                                                PAGING              Real,MB    31488
Disk      Busy%     KBPS     TPS KB-Read KB-Writ Faults      423    % Comp        41
Total     100.0   5027.1   335.0    63.7  4963.3 Steals        0    % Noncomp     29
                                                PgspIn         0    % Client      29
FileSystem          KBPS     TPS KB-Read KB-Writ PgspOut       0
Total               2.3K   139.5    78.1    2.2K PageIn       11    PAGING SPACE
                                                PageOut        4    Size,MB    32768
                                                Sios          16    % Used         0
                                                                    % Free       100
                                                NFS (calls/sec)
```

图 2-5　存储系统状态数据

当 Disk 的 Busy%列持续接近 100%时，表示当前系统可能存在 I/O 资源瓶颈，但是此指标不能作为判断的唯一标准，还需要通过 KBPS、TPS 等指标来综合判断。KBPS 代表磁盘的吞吐量，也就是每秒磁盘 I/O 的流量，即磁盘每秒钟读写数据的大小。TPS 这个指标尤其重要，它对应的是 IOPS，即 I/O Per Second，也就是每秒进行读写(I/O)操作的次数，它是衡量存储性能的一个重要指标。但此指标与实际的物理场景有很大关系，如转速为 15000 转的 FC 磁盘，随机读写的 TPS 理论值在 185。当发现某个磁盘的 TPS 值接近于这个理论值的时候，就表示此磁盘肯定存在 I/O 瓶颈。这时可以参考 2.1.5 节来进一步分析存储资源的运行情况。在 Disk 部分，可以获得每个磁盘的繁忙比例，多个磁盘的繁忙就会成为整个系统的性能瓶颈，这时应同时观察到 Wait 和 Waitqueue 持续不为 0，表示获得 CPU 资源的进程在等待 I/O 数据。如果同时观察到系统内存资源紧张，就应先解决物理内存资源不足的问题，因为系统内存资源紧张会导致大量的换页操作，产生大量的磁盘 I/O，从而引起磁盘繁忙。

图 2-6 中的框选部分显示了网络状态数据。

```
Topas Monitor for host:    ARAPPSQ                EVENTS/QUEUES     FILE/TTY
Wed Oct 10 11:04:59 2012   Interval:  2           Cswitch      375  Readch      676
                                                  Syscall     1935  Writech    1436
Kernel     0.2    |#                          |   Reads          2  Rawin         0
User       0.0    |#                          |   Writes         4  Ttyout      268
Wait       0.0    |                           |   Forks          0  Igets         0
Idle      99.8    |###########################|   Execs          0  Namei         1
                                                  Runqueue     0.0  Dirblk        0
Network  KBPS   I-Pack  O-Pack    KB-In  KB-Out   Waitqueue    0.0
lo0      11.8    109.0   109.0      5.9     5.9                     MEMORY
en0       1.0      9.5     1.5      0.6     0.4   PAGING            Real,MB   31488
                                                  Faults         0  % Comp     41
Disk     Busy%     KBPS     TPS KB-Read KB-Writ   Steals         0  % Noncomp  29
Total      0.0      0.0     0.0     0.0     0.0   PgspIn         0  % Client   29
                                                  PgspOut        0
FileSystem         KBPS     TPS KB-Read KB-Writ   PageIn         0  PAGING SPACE
Total               0.2     1.0     0.2     0.0   PageOut        0  Size,MB   32768
                                                  Sios           0  % Used      0
                                                                    % Free    100
```

图 2-6　网络状态数据

在 Network 部分，可以观察到每个网卡的通信数据量，结合网络带宽的硬件配置，就可以计算出当前网络通信带宽与硬件配置带宽的比例，例如千兆以太网的通信流量实际最高只能达到 85MB/S 左右。如果实际流量接近这个值，就表示网络通信部分很可能存在性能瓶颈。I-Pack 和 O-Pack 代表流入和流出的数据包的量，这两个值的高低，也会代表网卡的繁忙程度，比如千兆网卡每秒钟处理的数据包的数量在 20 万以下，而万兆网卡每秒钟处理的数据包的数量则可以达到 50 万左右。

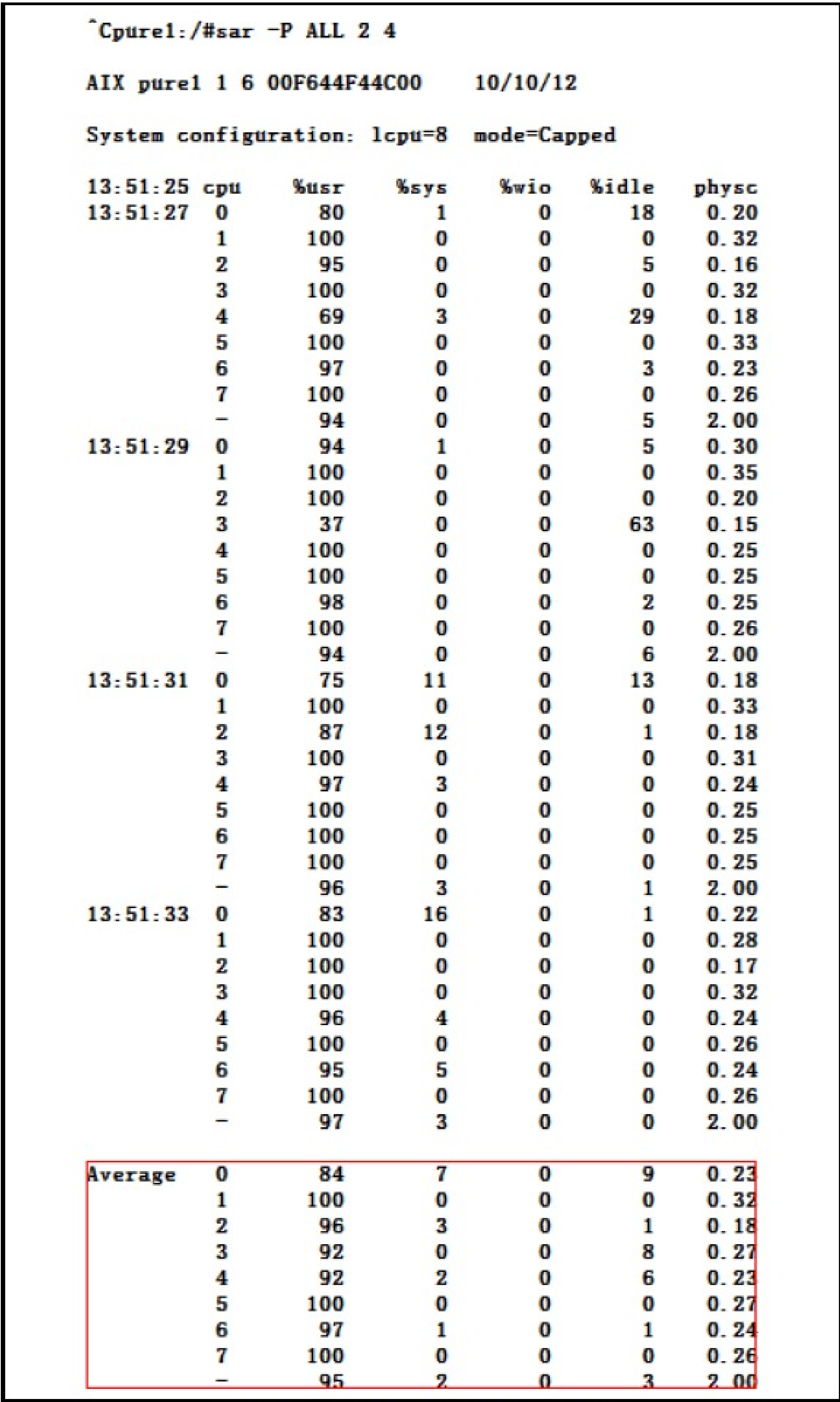
某些用户应用需要大量网络通信，所以网络通信数据量可以持续较高，但当系统不存在其他方面的性能瓶颈，同时网络通信数据带宽持续接近网络物理带宽时，就表示网络带宽已成为系统瓶颈，这时需要参考 2.2.6 节来进一步分析网络资源的运行情况。

如果在 topas 监控界面没有能很直观地观察到以上列出的性能问题，就表示系统当时基本运行正常。如果这时需要长期对性能情况进行跟踪，就可以在系统后台运行 nmon 工具来收集系统全天运行情况的数据并做进一步分析。因为系统性能问题与应用负载相关，应用负载在每天的不同时段可能发生变化，所以收集系统全天运行数据进行分析是监测系统性能状态的必要工作。在系统全天运行数据中识别出应用繁忙时段和繁忙的硬件资源，然后在相应时段使用系统性能监控工具，对系统运行状态进行详细分析。nmon 工具是实现这一需求的首选工具，它在 AIX 6.1 以及之后的版本中已经成为操作系统自带的工具。nmon 工具除了收集的指标很全之外，还有配套的工具集可以帮助完成一些后续的数据处理工作，比如可以使用这些工具将收集到的数据转换为图形化的 Excel 格式的报告，非常便于分析。关于此工具更详细的使用方法，有兴趣的读者可以到 IBM 的网站查询，本书篇幅有限不再进一步说明。

2.1.3 监控 CPU 性能

CPU 是整个系统的核心资源，通常也是系统的最繁忙部分，某些用户应用需要进行大量计算，所以 CPU 可能会保持繁忙。

通常，服务器都会配置多个 CPU，用 sar 命令可以收集每个 CPU 的使用情况。图 2-7 所示为 2 秒间隔收集 4 次 CPU 使用比例的数据，方框内的数据反映了 CPU 的平均使用情况。如果只有部分 CPU 繁忙，其他 CPU 保持空闲，可以调查应用运行特点，单线程的应用只能运行在单个 CPU 上，不能充分利用多个 CPU 的计算能力，所以尽量配置应用为多进(线)程、多并发的应用。



```
^Cpure1:/#sar -P ALL 2 4

AIX pure1 1 6 00F644F44C00    10/10/12

System configuration: lcpu=8  mode=Capped

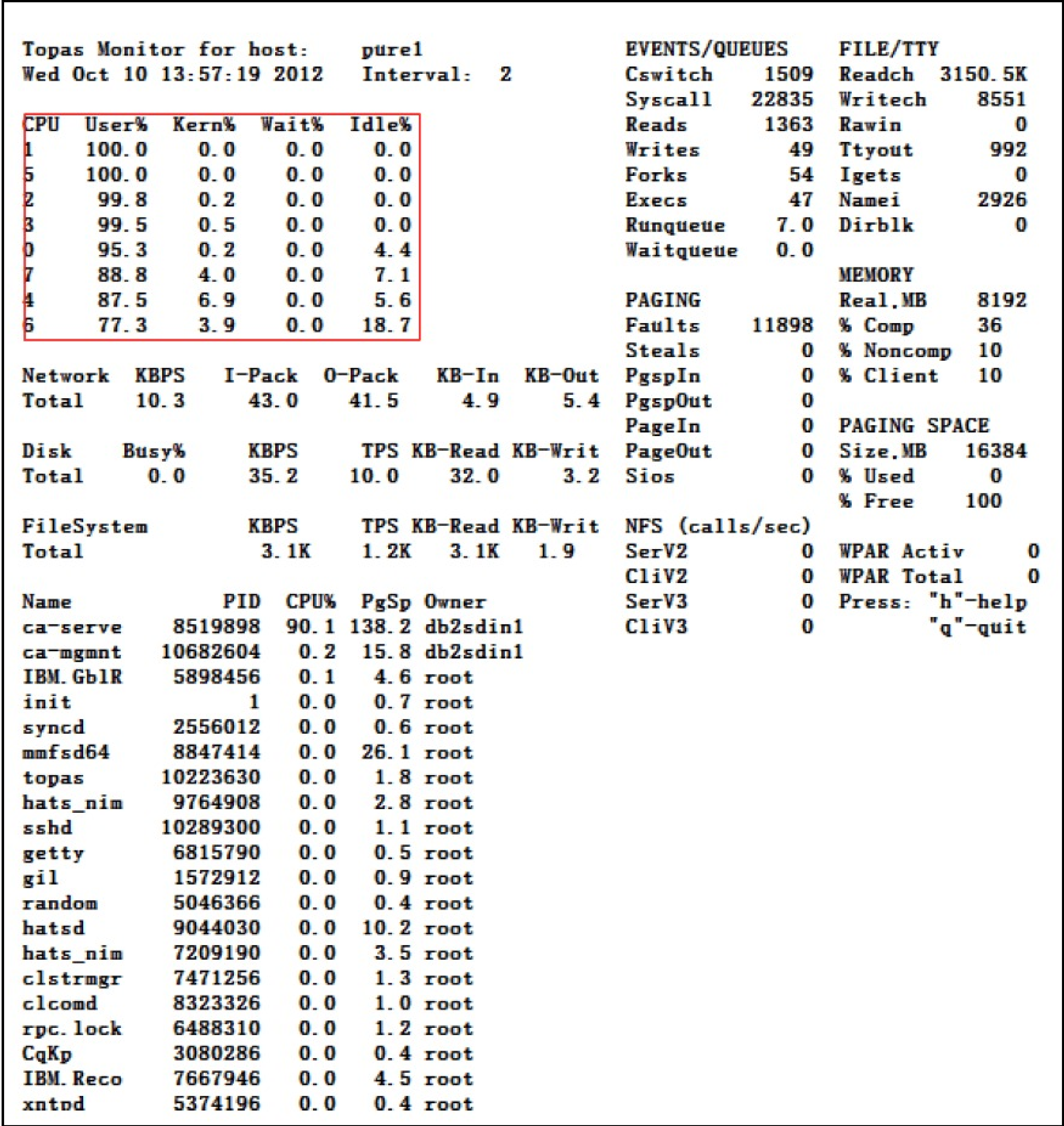
13:51:25 cpu    %usr    %sys    %wio   %idle   physc
13:51:27  0      80       1       0      18    0.20
          1     100       0       0       0    0.32
          2      95       0       0       5    0.16
          3     100       0       0       0    0.32
          4      69       3       0      29    0.18
          5     100       0       0       0    0.33
          6      97       0       0       3    0.23
          7     100       0       0       0    0.26
          -      94       0       0       5    2.00
13:51:29  0      94       1       0       5    0.30
          1     100       0       0       0    0.35
          2     100       0       0       0    0.20
          3      37       0       0      63    0.15
          4     100       0       0       0    0.25
          5     100       0       0       0    0.25
          6      98       0       0       2    0.25
          7     100       0       0       0    0.26
          -      94       0       0       6    2.00
13:51:31  0      75      11       0      13    0.18
          1     100       0       0       0    0.33
          2      87      12       0       1    0.18
          3     100       0       0       0    0.31
          4      97       3       0       0    0.24
          5     100       0       0       0    0.25
          6     100       0       0       0    0.25
          7     100       0       0       0    0.25
          -      96       3       0       1    2.00
13:51:33  0      83      16       0       1    0.22
          1     100       0       0       0    0.28
          2     100       0       0       0    0.17
          3     100       0       0       0    0.32
          4      96       4       0       0    0.24
          5     100       0       0       0    0.26
          6      95       5       0       0    0.24
          7     100       0       0       0    0.26
          -      97       3       0       0    2.00

Average   0      84       7       0       9    0.23
          1     100       0       0       0    0.32
          2      96       3       0       1    0.18
          3      92       0       0       8    0.27
          4      92       2       0       6    0.23
          5     100       0       0       0    0.27
          6      97       1       0       1    0.24
          7     100       0       0       0    0.26
          -      95       2       0       3    2.00
```

图 2-7　收集 CPU 使用情况

如果系统中 CPU 的总利用率(用户+系统)长时间持续大于 80%，可以认为系统中 CPU 可能是瓶颈。经验法则是：保持 CPU 利用率(大部分情况下)低于 80%，这样可以为处理突然增加的大量活动预留 CPU 的处理能力。如果 CPU 存在明显的瓶颈，就要找到造成瓶颈的原因并进行调优，找出原因的第一步就是定位占用 CPU 最多的进程，接着再找出进程中消耗 CPU 最多的程序或代码，也就是到底在 CPU 中执行了什么。

在 topas 监控界面，使用 c 命令可以切换到 CPU 视图，从而观察每个 CPU 的使用情况，如图 2-8 所示。



```
Topas Monitor for host:    pure1              EVENTS/QUEUES    FILE/TTY
Wed Oct 10 13:57:19 2012   Interval:  2       Cswitch    1509  Readch   3150.5K
                                              Syscall   22835  Writech     8551
CPU  User%  Kern%  Wait%  Idle%               Reads      1363  Rawin          0
1    100.0    0.0    0.0    0.0               Writes       49  Ttyout       992
5    100.0    0.0    0.0    0.0               Forks        54  Igets          0
2     99.8    0.2    0.0    0.0               Execs        47  Namei       2926
3     99.5    0.5    0.0    0.0               Runqueue    7.0  Dirblk         0
0     95.3    0.2    0.0    4.4               Waitqueue   0.0
7     88.8    4.0    0.0    7.1                                MEMORY
4     87.5    6.9    0.0    5.6               PAGING           Real,MB     8192
6     77.3    3.9    0.0   18.7               Faults    11898  % Comp        36
                                              Steals        0  % Noncomp     10
Network  KBPS   I-Pack  O-Pack   KB-In  KB-Out PgspIn       0  % Client      10
Total    10.3    43.0    41.5     4.9    5.4   PgspOut      0
                                              PageIn        0  PAGING SPACE
Disk     Busy%    KBPS     TPS KB-Read KB-Writ PageOut      0  Size,MB    16384
Total      0.0    35.2    10.0    32.0    3.2  Sios         0  % Used         0
                                                               % Free       100
FileSystem        KBPS     TPS KB-Read KB-Writ NFS (calls/sec)
Total              3.1K    1.2K    3.1K    1.9 SerV2        0  WPAR Activ     0
                                              CliV2         0  WPAR Total     0
Name            PID  CPU%  PgSp Owner         SerV3         0  Press: "h"-help
ca-serve    8519898  90.1 138.2 db2sdin1      CliV3         0         "q"-quit
ca-mgmnt   10682604   0.2  15.8 db2sdin1
IBM.GblR    5898456   0.1   4.6 root
init              1   0.0   0.7 root
syncd       2556012   0.0   0.6 root
mmfsd64     8847414   0.0  26.1 root
topas      10223630   0.0   1.8 root
hats_nim    9764908   0.0   2.8 root
sshd       10289300   0.0   1.1 root
getty       6815790   0.0   0.5 root
gil         1572912   0.0   0.9 root
random      5046366   0.0   0.4 root
hatsd       9044030   0.0  10.2 root
hats_nim    7209190   0.0   3.5 root
clstrmgr    7471256   0.0   1.3 root
clcomd      8323326   0.0   1.0 root
rpc.lock    6488310   0.0   1.2 root
CqKp        3080286   0.0   0.4 root
IBM.Reco    7667946   0.0   4.5 root
xntpd       5374196   0.0   0.4 root
```

图 2-8　CPU 视图

使用 vmstat 命令也可以收集 CPU 使用情况的数据，如图 2-9 所示。

```
pure1:/#vmstat -w 1 6

System configuration: lcpu=8 mem=8192MB

 kthr          memory                          page                              faults           cpu
------- --------------------- ------------------------------------ ------------------ -----------
  r   b        avm        fre    re    pi    po    fr     sr    cy    in     sy    cs us sy id wa
 12   0     746257    1126566     0     0     0     0      0     0    43  37055  1514 94  3  3  0
  7   0     746257    1126566     0     0     0     0      0     0    10    583  1102 94  0  6  0
  7   0     746257    1126566     0     0     0     0      0     0     6    558  1016 94  0  6  0
  7   0     746257    1126566     0     0     0     0      0     0     8    577  1027 94  0  6  0
  7   0     746257    1126566     0     0     0     0      0     0     7    613  1112 94  0  6  0
  7   0     748055    1124768     0     0     0     0      0     0     8   3611  1175 94  1  5  0
```

图 2-9　使用 vmstat 命令

当已确定 CPU 繁忙时，需要确定哪些进程正在消耗 CPU 资源，在 topas 监控界面，使用 P 命令可以切换到进程视图，如图 2-10 所示。移动光标到 CPU%列，对进程按照 CPU 使用量降序排列，大部分的 CPU 资源应该消耗在系统的应用进程上。如果出现非核心应用进程占用大量 CPU 资源，就应该分析该进程的作用和产生原因，以避免出现 CPU 资源浪费在与业务无关的进程上的情况。

```
Topas Monitor for host:    pure1        Interval:   2    Wed Oct 10 14:06:46 2012

                                 DATA  TEXT  PAGE                PGFAULTS
USER          PID    PPID PRI NI  RES   RES SPACE     TIME CPU%  I/O  OTH COMMAND
db2sdin18519898 10682604  60 20 35387     4 35387 18663:47 92.2    0    0 ca-serve
db2sdin110682604 11075822  60 20  4048    29  4048    21:15  0.1    0    0 ca-mgmnt
root      9699572 10879136  58 41   482   141   482   0:00  0.0    0    0 topas
root      5898456 5636270  60 20  1168    87  1168   9:01  0.0    0  223 IBM.GblR
root       983070       0  60 41   112     0   112   1:34  0.0    0    0 xmgc
root      8847414 6225958  40 41  6686  2013  6686   3:00  0.0    0    0 mmfsd64
root      9961694 5636270  60 20   932   479   932   0:14  0.0    0    0 IBM.Stor
root      6815790       1  60 20   140    21   140   1:15  0.0    0    0 getty
root      9764908 9044030  38 41   719    83   719   1:09  0.0    0    0 hats_nim
root      5046366       1  60 20   112     0   112   0:42  0.0    0    0 random
root      1572912       0  37 41   240     0   240   0:46  0.0    0    0 gil
root      8716372 5308630  60 20   280   155   280   0:00  0.0    0    0 sshd
root      7209190 9044030  38 41   905    83   905   0:27  0.0    0    0 hats_nim
root      7471256 5636270  39 20   336   183   336   0:31  0.0    0    0 clstrmgr    R
root      8323326 5636270  60 20   255   128   255   0:15  0.0    0    0 clcomd
root      9044030 5636270  38 41  2605   335  2605   0:21  0.0    0    0 hatsd
root      6488310       1  60 20   304     0   304   0:10  0.0    0    0 rpc.lock
root      5374196 5636270  48  8    99    67    99   0:05  0.0    0    0 xntpd
root      7667946 5636270  60 20  1150   566  1150   0:16  0.0    0    0 IBM.Reco
```

图 2-10　进程视图

要进一步分析进程和系统命令对 CPU 资源的使用情况，可以使用 tprof -usekj -x sleep 9 命令，该命令收集 9 秒的系统数据并生成图 2-11 所示报告，收集数据时间长短可调整。需要注意的是：此命令采用 AIX 底层的 trace 功能完成信息的收集，需要消耗一部分 CPU 的资源，在系统的 CPU 已经很繁忙的情况下使用此命令收集信息时，会相应加重系统的负担。

该命令会自动创建结果报告，文件的名称默认为 sleep.prof。图 2-11 显示的只是此报告的开始部分。

```
pure1:/#cat sleep.prof
Configuration information
=========================
System: AIX 6.1 Node: pure1 Machine: 00F644F44C00
Tprof command was:
    tprof -usekj -x sleep 9
Trace command was:
    /usr/bin/trace -ad -M -L 772494540 -T 500000 -j 00A,001,002,003,38F,005,006,134,
Total Samples = 7213
Traced Time = 9.02s (out of a total execution time of 9.02s)
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Process                              Freq  Total Kernel   User Shared  Other   Java
=======                              ====  ===== ======   ==== ======  =====   ====
ca-server                               7  87.12   0.00   0.00  87.12   0.00   0.00
wait                                    5  11.56  11.56   0.00   0.00   0.00   0.00
e/db2sdin1/sqllib/adm/db2rocme          2   0.24   0.21   0.00   0.03   0.00   0.00
-ksh                                   10   0.19   0.19   0.00   0.00   0.00   0.00
/bin/ksh                                6   0.12   0.11   0.01   0.00   0.00   0.00
/usr/bin/echo                           8   0.11   0.11   0.00   0.00   0.00   0.00
db2rocm                                 2   0.11   0.10   0.00   0.00   0.01   0.00
/usr/bin/awk                            8   0.11   0.10   0.01   0.00   0.00   0.00
ExecCCommand                            5   0.10   0.10   0.00   0.00   0.00   0.00
ca-mgmnt-lwd                            4   0.10   0.10   0.00   0.00   0.00   0.00
/usr/bin/sed                            5   0.07   0.07   0.00   0.00   0.00   0.00
/usr/bin/tprof                          1   0.04   0.00   0.03   0.01   0.00   0.00
IBM.GblResRMd                           2   0.03   0.03   0.00   0.00   0.00   0.00
/usr/bin/trcstop                        1   0.01   0.00   0.00   0.01   0.00   0.00
/usr/bin/ksh                            1   0.01   0.01   0.00   0.00   0.00   0.00
/usr/bin/grep                           1   0.01   0.01   0.00   0.00   0.00   0.00
/usr/bin/cat                            1   0.01   0.01   0.00   0.00   0.00   0.00
/usr/bin/sh                             1   0.01   0.01   0.00   0.00   0.00   0.00
xmgc                                    1   0.01   0.01   0.00   0.00   0.00   0.00
/usr/bin/hostname                       1   0.01   0.01   0.00   0.00   0.00   0.00
=======                              ====  ===== ======   ==== ======  =====   ====
Total                                  72 100.00  12.75   0.06  87.18   0.01   0.00

Process                 PID      TID  Total Kernel   User Shared  Other   Java
=======                 ===      ===  ===== ======   ==== ======  =====   ====
ca-server           8519898 29687991  12.51   0.00   0.00  12.51   0.00   0.00
ca-server           8519898 47186157  12.51   0.00   0.00  12.51   0.00   0.00
ca-server           8519898 10354911  12.51   0.00   0.00  12.51   0.00   0.00
ca-server           8519898 44695701  12.49   0.00   0.00  12.49   0.00   0.00
ca-server           8519898 25821349  12.44   0.00   0.00  12.44   0.00   0.00
ca-server           8519898 10747953  12.41   0.00   0.00  12.41   0.00   0.00
ca-server           8519898 40435879  12.27   0.00   0.00  12.27   0.00   0.00
```

图 2-11　收集系统数据

通过图 2-12 中所示的命令，可以获得占用 CPU 资源较多的函数调用，为代码级的性能调整提供线索。

```
  Total % For All Processes (USER) = 0.06

User Process                                                   %
=============                                                ======
/usr/bin/tprof                                                 0.03
/bin/ksh                                                       0.01
/usr/bin/awk                                                   0.01

  Profile: /usr/bin/tprof

  Total % For All Processes (/usr/bin/tprof) = 0.03

Subroutine                                 %    Source
==========                              ====== ======
.trc_kern_prof                            0.01 /usr/bin/tprof
.update_tidtbl                            0.01 /usr/bin/tprof

  Profile: /bin/ksh

  Total % For All Processes (/bin/ksh) = 0.01

Subroutine                                 %    Source
==========                              ====== ======
.wcslen                                   0.01 glink.s

  Profile: /usr/bin/awk

  Total % For All Processes (/usr/bin/awk) = 0.01

Subroutine                                 %    Source
==========                              ====== ======
.print                                    0.01 /usr/bin/awk

  Total % For All Processes (KERNEL) = 12.74

Subroutine                                 %    Source
==========                              ====== ======
h_cede_end_point                         11.56 hcalls.s
pcs_glue                                  0.24 vmvcs.s
.unlock_enable_mem                        0.10 low.s
hk_restore_userkeys                       0.07 64/skeys.s
.ld_resolveredo                           0.04 nel/ldr/ld_symbols.c
.v_lookup_mpss                            0.03 ernel/vmm/v_lookup.c
```

图 2-12　其他命令

2.1.4　监控内存使用

当监控内存时，应确保系统中不存在频繁的内存页的换入(pi-pagingspace pagein)和换出(po-pagingspace pageout)，并且系统的空闲内存在 5%左右。

当系统持续进行换页操作时，便可以确定物理内存资源紧张或者内存资源分配不合理。这时首先要确定，通过调整虚拟内存管理器内核参数来重新分配内存资源能否解决物理内存资源不足的问题。使用 svmon 命令可以获得物理内存使用的总体信息，如图 2-13 所示。内存的 size 值给出了系统装配的物理内存页面数，virtual 值给出了系统工作所需的页面数。如果 virtual 值大于 size 值，就表示物理内存资源不足，换页不可避免，此时只

能在硬件上升级物理内存或配置用户应用，或者通过减小应用对内存的需求量来解决此性能瓶颈。

```
pure1:/#svmon -G
               size       inuse        free         pin     virtual   mmode
memory      2097152      967649     1129503      537933      741636     Ded
pg space    4194304        3692

               work        pers        clnt       other
pin          455876           0       12185       69872
in use       741636          30      225983

PageSize   PoolSize       inuse        pgsp         pin     virtual
s    4 KB         -      462609        3692      135901      236596
m   64 KB         -       31565           0       25127       31565
```

图 2-13　物理内存使用的总体信息

vmstat 命令可用来收集内存使用数据和换页数据，如图 2-14 所示。内存的 fre 列表示空闲的物理内存页面数；内存的 avm 列表示系统工作所需的页面数。如果此页面数对应的容量大于系统物理内存容量，就表示物理内存不足，换页不可避免。需要采取措施来处理此瓶颈。

```
pure1:/#vmstat -w 1 6

System configuration: lcpu=8 mem=8192MB

 kthr          memory                         page                       faults           cpu
------- --------------------- ------------------------------------ ------------------ -----------
  r   b        avm        fre    re    pi    po    fr     sr    cy    in     sy    cs us sy id wa
  8   0     743021    1128113     0     0     0     0      0     0    19  25554  1547 97  3  0  0
 10   0     741821    1129313     0     0     0     0      0     0    25  27111  1506 97  3  0  0
 10   0     741499    1129634     0     0     0     0      0     0    17  17794  1463 96  2  2  0
  7   0     741499    1129634     0     0     0     0      0     0    13   6056  1210 94  1  5  0
  8   0     741887    1129245     0     0     0     0      0     0     6  10875  1371 94  1  4  0
  7   0     741523    1129610     0     0     0     0      0     0    42  21752  1492 94  2  5  0
```

图 2-14　使用 vmstat 命令

使用 lsps 命令可以查看换页空间(Paging Space)的使用情况，如图 2-15 所示。换页空间使用率高并持续增加是物理内存资源不足的表现。如果在这种情况下还不能及时采取措施以改善内存的使用，那么最终可能会导致操作系统宕机。

```
pure1:/#lsps -a
Page Space      Physical Volume   Volume Group Size %Used Active  Auto  Type Chksum
hd6             hdisk0            rootvg       16384MB    80   yes   yes    lv     0
```

图 2-15　使用 lsps 命令

使用 vmstat -v 命令可获得当前系统物理内存中计算内存和文件缓存的配置参数，以及当前的使用情况，如图 2-16 所示。

```
pure1:/#vmstat -v
              2097152 memory pages
              2018784 lruable pages
              1129061 free pages
                    1 memory pools
               538189 pinned pages
                 80.0 maxpin percentage
                  3.0 minperm percentage
                 90.0 maxperm percentage
                 10.5 numperm percentage
               213515 file pages
                  0.0 compressed percentage
                    0 compressed pages
                 10.5 numclient percentage
                 90.0 maxclient percentage
               213485 client pages
                    0 remote pageouts scheduled
                    0 pending disk I/Os blocked with no pbuf
                    0 paging space I/Os blocked with no psbuf
                 2228 filesystem I/Os blocked with no fsbuf
                 1348 client filesystem I/Os blocked with no fsbuf
                    0 external pager filesystem I/Os blocked with no fsbuf
                 36.0 percentage of memory used for computational pages
```

图 2-16　使用 vmstat-v 命令

当系统工作内存不足时，下面的命令可以限制文件缓存对物理内存的占用比例：

```
#vmo -p -o lru_file_repage=0 -o maxclient%=90 -o maxperm%=90 -o minperm%=3
#vmo -p -o strict_maxclient=1 -o strict_maxclient=0
```

- 使用 svmon -S 命令可以获得所有虚拟内存段对物理内存的使用情况。
- 使用 svmon -S -s 命令可以获得所有系统内核内存段对物理内存的使用情况。
- 使用 svmon -U 命令可以列出所有用户对物理内存的使用情况。
- 使用 svmon -P 命令可以列出所有进程对物理内存的使用情况。
- 使用 svmon -P pid 命令可以显示特定进程对物理内存的使用情况。
- 使用 svmon -C 命令可以显示由特定命令构成的所有进程对物理内存的使用情况。

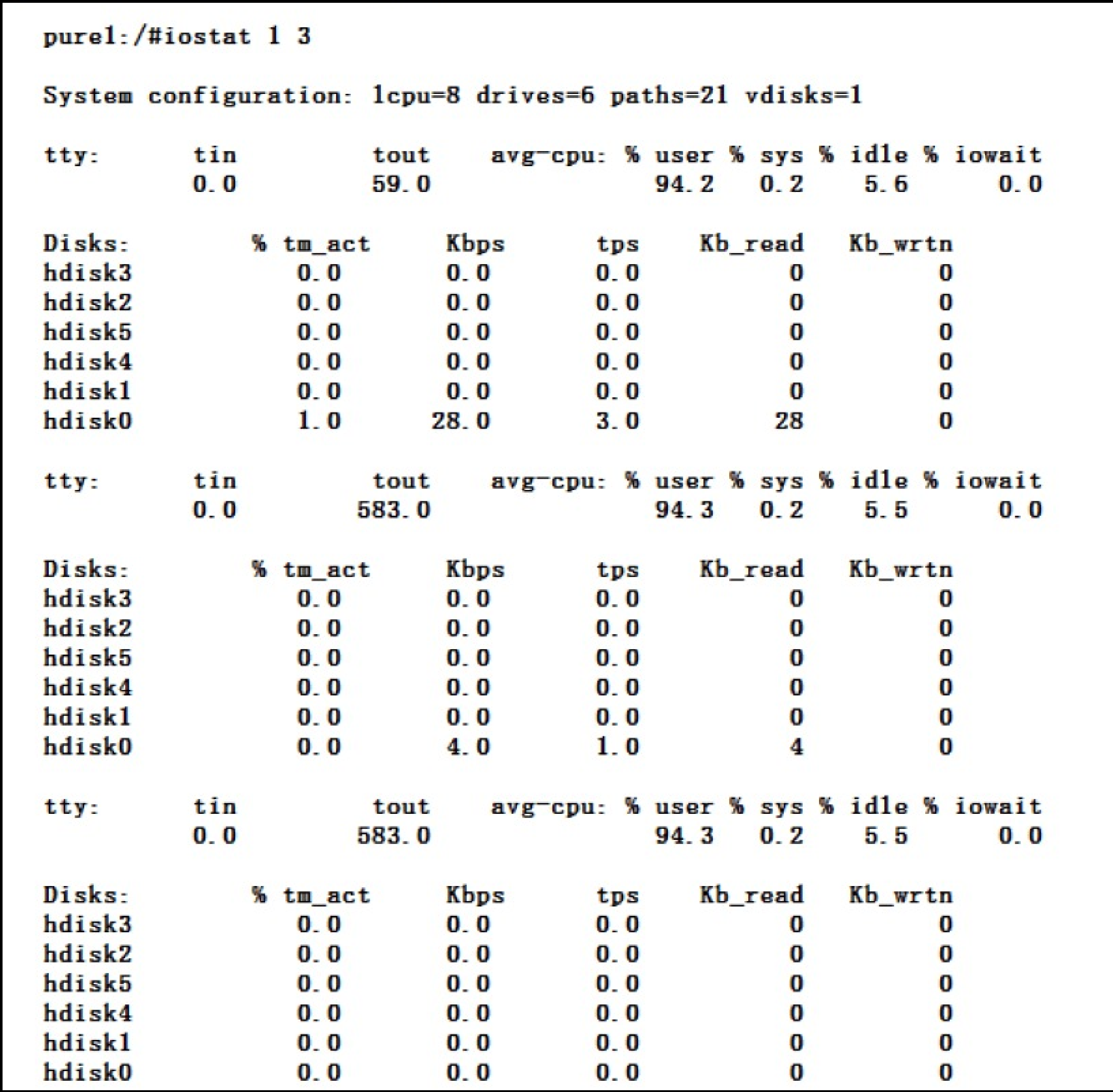
请参看 AIX 虚拟内存管理文档，以了解以上命令的具体使用方法。

2.1.5　监控存储系统状态

如果系统中磁盘 I/O wait 长时间持续大于 25%，可以认为系统可能存在 I/O 瓶颈。如

果存在 I/O 瓶颈，就需要采取多种措施进行调优，如调整 I/O 资源的设置、调整 I/O 资源的分布等手段，除了这些方法，还可以通过调整应用或数据库层次以减少不必要的 I/O 访问，最终提高系统的性能。

当观测到系统 I/O 保持繁忙时，可以先使用 topas 查看存储的繁忙程度，监控 KBPS、TPS 等指标，然后使用 iostat 命令收集磁盘使用数据，如图 2-17 所示。该命令可以收集磁盘带宽的使用率和磁盘 I/O 数据吞吐量，可以大致估计系统数据吞吐量与应用负载是否相匹配，避免与业务无关的大量 I/O 操作。



```
pure1:/#iostat 1 3

System configuration: lcpu=8 drives=6 paths=21 vdisks=1

tty:       tin         tout    avg-cpu: % user % sys % idle % iowait
           0.0         59.0              94.2    0.2    5.6      0.0

Disks:          % tm_act     Kbps      tps    Kb_read   Kb_wrtn
hdisk3             0.0       0.0       0.0          0         0
hdisk2             0.0       0.0       0.0          0         0
hdisk5             0.0       0.0       0.0          0         0
hdisk4             0.0       0.0       0.0          0         0
hdisk1             0.0       0.0       0.0          0         0
hdisk0             1.0      28.0       3.0         28         0

tty:       tin         tout    avg-cpu: % user % sys % idle % iowait
           0.0        583.0              94.3    0.2    5.5      0.0

Disks:          % tm_act     Kbps      tps    Kb_read   Kb_wrtn
hdisk3             0.0       0.0       0.0          0         0
hdisk2             0.0       0.0       0.0          0         0
hdisk5             0.0       0.0       0.0          0         0
hdisk4             0.0       0.0       0.0          0         0
hdisk1             0.0       0.0       0.0          0         0
hdisk0             0.0       4.0       1.0          4         0

tty:       tin         tout    avg-cpu: % user % sys % idle % iowait
           0.0        583.0              94.3    0.2    5.5      0.0

Disks:          % tm_act     Kbps      tps    Kb_read   Kb_wrtn
hdisk3             0.0       0.0       0.0          0         0
hdisk2             0.0       0.0       0.0          0         0
hdisk5             0.0       0.0       0.0          0         0
hdisk4             0.0       0.0       0.0          0         0
hdisk1             0.0       0.0       0.0          0         0
hdisk0             0.0       0.0       0.0          0         0
```

图 2-17　收集磁盘使用数据

使用 df 命令可以显示文件系统剩余容量，如图 2-18 所示。文件系统剩余容量过低或出现容量满时，写磁盘失败会导致很严重的系统问题。

```
pure1:/#df -sg
Filesystem    GB blocks      Free* %Used     Iused %Iused Mounted on
/dev/hd4           6.00       5.32   12%      9130     1% /
/dev/hd2          10.00       6.22   38%     88819     6% /usr
/dev/hd9var        5.50       5.24    5%      3645     1% /var
/dev/hd3          10.00       9.01   10%      3158     1% /tmp
/dev/hd1          10.50      10.50    1%        11     1% /home
/dev/hd11admin      0.50       0.50    1%         5     1% /admin
/proc                 -          -     -         -      - /proc
/dev/hd10opt      10.50       6.06   43%     15510     2% /opt
/dev/livedump      0.50       0.50    1%         4     1% /var/adm/ras/livedump
/dev/lv00          0.50       0.48    4%        20     1% /var/adm/csd
/dev/lvcmbc_admin      10.00       9.77    3%        16     1% /cmbc_admin
/dev/db2data     300.00     135.01   55%      4284     2% /db2data
/dev/db2fs1      100.00      98.77    2%      4532     4% /db2sd_20120908150547
```

图 2-18　显示文件系统剩余容量

dd 命令可以与 time 命令配合来测试 I/O 的顺序读、写带宽。存储带宽测试工作需要选择系统空闲时段进行，从而避免影响实际系统的性能。获得存储带宽后，就可以与实际的业务需求进行比较，判断存储带宽是否满足业务负载需求。但需要注意的是，此类型的 I/O 测试的 I/O 是顺序 I/O，而实际的场景中往往是随机 I/O，随机 I/O 的访问效率要远远低于顺序 I/O。

这里简单补充一下顺序 I/O 和随机 I/O 的基本概念。顺序 I/O 指的是本次 I/O 给出的初始扇区地址和上一次 I/O 的结束扇区地址完全连续或相隔不多。反之，如果相差很大，就算作一次随机 I/O。顺序 I/O 比随机 I/O 效率高的原因是，在做顺序 I/O 的时候，磁头几乎不用换道，或者换道的时间很短；而对于随机 I/O，如果这种 I/O 很多的话，会导致磁头不停地换道，造成效率极大降低。下面列出了典型的 FC 磁盘的相关指标，例如转速为 15000 转、容量为 148GB 的磁盘的指标如下：

- 顺序 I/O 速率：88MB/s
- 顺序 I/O 的 IOPS：1421
- 随机 I/O 速率：12MB/s
- 随机 I/O 的 IOPS：185

由此我们看到对于同样的磁盘，I/O 访问的类型不一样，I/O 的能力区别也会非常大，因此，我们在评估 I/O 瓶颈时需要根据实际情况进行分析和判断。

2.1.6　监控网络状态

网络成为性能瓶颈的场景并不多见，一般多发生在多分区数据库的环境下，在网络负载较高的情况下，对这方面进行某些调优也可以提高性能。如果系统的 CPU 和 I/O 利用率都很低，那么可以进一步观察、分析网络是否可能是性能的瓶颈。在多分区数据库(DPF)中，如果分区策略产生了一些非合并连接，那么可能导致最严重的性能下降，而且瓶颈很

可能是 I/O 造成的，因为对于 DPF 数据库，分区之间可能会出现大量的数据传输，从而造成网络成为性能的瓶颈。

使用 netstat 命令监控网络通信状况，如图 2-19 所示。如果 errs 列持续不为 0，就表示网络设备存在故障，需要定位具体原因来解决。

```
pure1:/#netstat 2
    input    (ib0)       output             input    (Total)     output
  packets   errs  packets   errs colls  packets   errs  packets   errs colls
  4512526      0  4326865      0     0 62140946      0 57250555      0     0
        3      0        3      0     0       27      0       25      0     0
        2      0        2      0     0       16      0       12      0     0
        3      0        3      0     0       20      0       15      0     0
        3      0        2      0     0       17      0       14      0     0
        3      0        3      0     0       86      0       78      0     0
        2      0        2      0     0       19      0       16      0     0
        3      0        3      0     0       26      0       24      0     0
        2      0        2      0     0       14      0       11      0     0
```

图 2-19　监控网络通信状况

使用 topas 命令可以查看网络通信数据的实时吞吐量，观察网络输入数据量和输出数据量是否与业务量相符，如图 2-20 所示。按照之前的分析，如果单块千兆网卡不满足性能要求时，可以更换成万兆网卡或通过双网卡绑定方式来增加网络吞吐量。

```
Network   KBPS   I-Pack  O-Pack   KB-In  KB-Out
en0        8.9    12.0    10.5     6.9     2.0
ib0        0.3     1.0     1.5     0.1     0.2
lo0        0.1     1.0     1.0     0.0     0.0
```

图 2-20　查看网络的实时吞吐量

在 AIX 系统中使用 no 命令可以调整网络通信相关的内核参数。

2.2　操作系统性能优化

在进行操作系统性能优化工作前，您需要了解应用系统所有的层次，因为它们分别会以不同的方式对性能产生影响。在第一次设置您的系统时，对于磁盘的配置，可以从底层(物理层)开始，然后是设备层、逻辑卷、文件系统、文件和应用程序。规划您的物理存储环境是非常重要的，这一点无论怎样强调都不为过。关于存储的 IO 设计我们在 2.5 节以后详细讲述，这涉及确定 RAID 级别、存储规划等。

如前所述，与任何可调整的 I/O 参数相比，糟糕的数据布局和 I/O 访问方式将会给 I/O 性能带来更大的影响。查看 I/O 栈可以帮助您理解这一点，因为逻辑卷管理器(Logical Volume Manager，LVM)和磁盘的分布情况要比优化参数(ioo 和 vmo)更接近于底层。

从图 2-21 可以看出，物理 I/O 组件与逻辑磁盘及其应用程序 I/O 之间非常紧密地关联在一起，这也正是通常将其称为 AIX I/O 栈的原因。

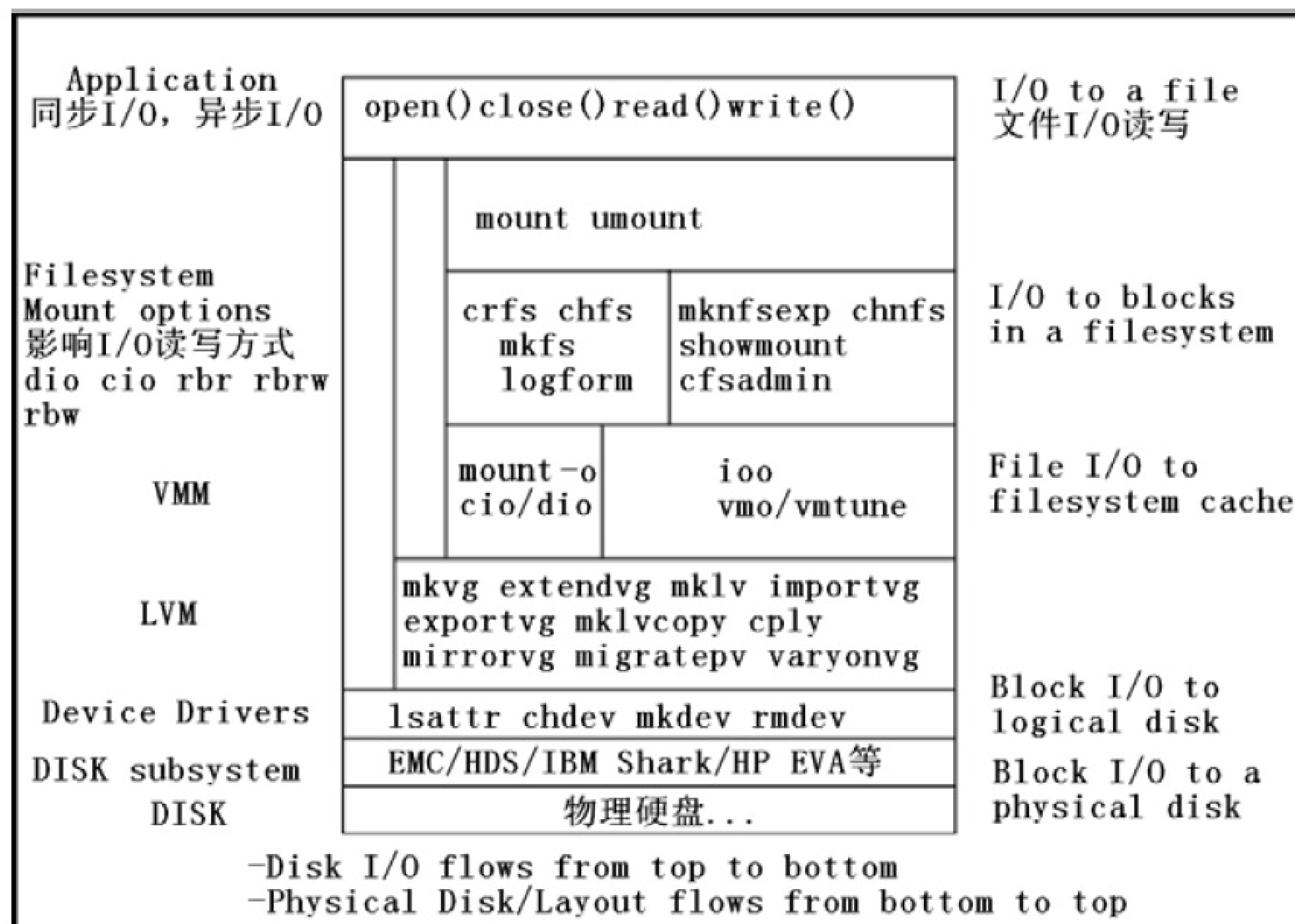


图 2-21　AIX I/O 栈

2.2.1　直接 I/O 和并发 I/O

操作系统和文件系统对 I/O 的控制存在多种方式，不同的 I/O 方式对数据库的 I/O 性能影响也不同。

1. 直接 I/O

那么，什么是直接 I/O(Direct I/O，DIO)呢？直接 I/O 在 AIX Version 4.3 中被首次引入，这种 I/O 方法不经过虚拟内存管理器(Virtual Memory Manager，VMM)，而是从应用程序的缓冲区直接与磁盘进行数据传输。在某些特定的场景，使用这种技术时可以提高 I/O 的性能。例如，那些具有糟糕缓存使用率的文件，是使用直接 I/O 的很好的候选对象。直接 I/O 还可以提高那些使用同步写入操作的应用程序的效率，因为这些写入操作的对象是磁盘。正常情况下，需要将磁盘的数据复制到文件系统缓冲区缓存，然后在将数据复制到应用程序的缓冲区时就会出现多余的副本。直接 I/O 降低了 CPU 使用率，这是因为减少了一次数据在内存中的复制。直接 I/O 的主要性能开销是，尽管可以降低 CPU 使用率，但却可能导致进程长时间等待较小请求的完成。请注意，这种方式适用于在磁盘上具有持久存储位置

的持久段文件。当使用 JFS2 不通过直接 I/O 访问文件时，文件作为本地页面缓存，并且将数据复制到 RAM 中。直接 I/O 可以在许多方面为您提供与使用原始逻辑卷(通常也可以称之为裸设备，raw device)类似的性能，同时还保留了使用 JFS 文件系统的优点(例如易于管理)。

2. 并发 I/O

并发 I/O(Concurrent I/O，CIO)又如何呢？并发 I/O 在 AIX Version 5.2 中被首次引入，首先，并发 I/O 包含了直接 I/O 相关联的所有功能和优势。使用标准的直接 I/O，会对索引节点(与文件相关联的数据结构)加锁，以防止出现多个线程试图同时更改某个文件的内容的情况。并发 I/O 绕过了索引节点锁，这样可以允许多个线程并发地读写相同文件的数据。这是因为 JFS2 在实现时使用了写独占(write-exclusive)索引节点锁，允许多个用户同时读取相同的文件。可以想象，直接 I/O 可能使得不断地从相同文件读取数据的数据库产生很大的问题。并发 I/O 则解决了这一问题，这正是该特性主要用于关系数据库的原因。与直接 I/O 类似，您可以通过 open 系统调用或者通过装入文件系统来实现这种方式，如下所示：

```
# mount -o cio /sapr3/prod
```

当您使用这个命令装入文件系统时，其中所有的文件都使用并发 I/O。与使用直接 I/O 相比，并发 I/O 几乎可以提供使用原始逻辑卷的所有优点，同时保持文件系统的易管理性。请注意，您不能对 JFS(只能对 JFS2)使用并发 I/O。而且，对于那些受益于文件系统预读功能或者较高缓冲区缓存命中率的应用程序，可能会出现性能的降低。因此使用 CIO 时需要确认应用确实能够支持这种 I/O 方式，并且能够从中获益，尤其是不能中途再修改为 CIO 方式，这样风险会很高。

2.2.2　异步 I/O 和同步 I/O

通常，用得比较多的 I/O 模型是同步 I/O(Synchronous I/O)。在这种模式下，当请求发出之后，应用程序就会被阻塞，直到请求返回为止。这种模式的最大好处就是调用应用程序在等待 I/O 请求完成时不需要使用 CPU 资源，而且这种方式也能满足必须完成特定的 I/O 才能执行后续工作的应用需求。但是，对于一些强调高响应速度的 OLTP 应用来说，希望这种等待时间越短越好，此时就该考虑采用异步 I/O(Asynchronous I/O)模式了。

异步 I/O 是什么情况呢？同步 I/O 和异步 I/O 的区别是应用程序是否等待 I/O 完成后再开始后续处理。其中，同步 I/O 指的是应用程序必须等待 I/O 处理完以后才能开始后续处

理，而异步 I/O 正好相反，应用程序不需要等 I/O 处理完，只要在发出了 I/O 请求之后就可以开始后续处理了，所以异步 I/O 允许应用程序在发出了 I/O 请求的同时继续处理其他工作。正确地使用异步 I/O 可以极大地提高 I/O 子系统读写操作的性能。这种方式能够提高性能是因为允许 I/O 和应用程序处理同时运行。对于数据库环境，启用异步 I/O 的确可以起到提高性能的作用。

异步 I/O 可以使需要大量读写的 DB2 线程(如 db2pfchr、db2pclnr 线程)将 I/O 请求队列化，以充分利用硬件的 I/O 带宽，从而使它们能在最大程度上实现并行处理。异步 I/O 还可以使那些需要进行大量计算的操作(如排序)，在它们发出 I/O 请求前预先从磁盘取出数据，使 I/O 和计算并行处理。

我们如何监控异步 I/O 服务器的使用率呢？iostat 和 nmon 命令都可以监视异步 I/O 服务器的使用率。确定您的系统中配置的异步 I/O 服务器数量的标准命令是：

```
pstat -a | egrep ' aioserver' | wc -l
```

iostat -A 命令可以报告异步 I/O 统计数据：

```
# iostat -A
System configuration: lcpu=2 drives=3 ent=0.60 paths=4 vdisks=4
aio: avgc avfc maxgc maxfc maxreqs avg-cpu: % user % sys % idle % iowait physc
% entc
  0    0    32    0      4096            6.4     8.0     85.4     0.2     0.1     16.0
Disks:          % tm      act       Kbps       tps      Kb read    Kb wrtn
hdisk0            0.5     2.0        0.5        0          4
hdisk1            1.0     5.9        1.5        8          4
hdisk2            0.0     0.0        0.0        0          0
```

所有这些内容究竟表示什么呢？

- avgc：报告了在您所指定的时间间隔内平均每秒的全局异步 I/O 请求数量。
- avfc：报告了在您所指定的时间间隔内平均每秒的快速路径请求数量。
- maxgc：报告了从上次获取该值以来的最大全局异步 I/O 请求。
- maxfc：报告了从上次获取该值以来的最大快速路径请求计数。
- maxreqs：这是所允许的最大异步 I/O 请求数。

在 AIX 6.1 及以后的版本中，异步 I/O 的使用和设置被大大简化了，一般无须进行设置，异步 I/O 会自动启动并且会自动调整异步 I/O 服务的数量。

2.2.3 minpout 和 maxpout

当系统内有其他应用在做大量的 I/O 操作时，用户可能会碰到诸如交互性能受到严重影响等问题。I/O 处理速率调整是 AIX 的一项特性，它可以防止因使用大量磁盘 I/O 的应用程序而使得 CPU 和磁盘超载。正确地使用磁盘 I/O 处理速率调整，可以帮助防止因生成大量输出的程序而使系统的 I/O 阻塞，并导致系统性能降低。优化 maxpout 和 minpout 可以帮助防止对文件执行顺序写入操作的线程占用过多的系统资源。

建议高端存储：minpout 设置成 512，maxpout 设置成 1024。低端存储：minpout 设置成 24，maxpout 设置成 33。

2.2.4 文件系统和裸设备

我们知道，内存的读写效率比磁盘高近万倍。因此，数据库通常会在内存中开辟一片区域，称为 Buffer Pool(或 Buffer Cache)，使数据的读写尽量在该部分内存中完成。同样，在文件系统中，操作系统为了提高读写效率，也会为文件系统开辟一块 Buffer 用于缓存最近被访问过的文件。这样，数据库中的数据很有可能会被缓存两次。为了避免两次缓存造成的内存和 CPU 浪费，所以不希望文件系统进行缓存。解决的方法有多种，其中一种(尤其是在没有 CIO 技术之前)是采用裸设备(Raw Device)作为数据文件的存储设备。裸设备，又称为裸分区(Raw Partition)，是指没有被加载(Mount)到操作系统的文件系统上的磁盘分区，可通过字符设备驱动来访问。裸设备的 I/O 读写不由操作系统控制，而是由应用程序(如数据库)直接控制。

1. 裸设备的优点

1) 由于屏蔽了文件系统缓冲区而进行直接读写，从而具有更好的性能。对硬盘的直接读写就意味着取消了硬盘与文件系统的同步需求。这一点对于纯 OLTP 系统非常有用，因为在这种系统中，读写的随机性非常大，以至于一旦数据被读写之后，它们在今后较长的一段时间内都不会得到再次使用。除了 OLTP，裸设备还能够从以下几个方面改善 OLAP 应用程序的性能：

- 排序：对于 OLAP 环境中大量存在的排序需求，裸设备所提供的直接写功能也非常有用，因为对临时表空间的写动作速度更快。
- 顺序访问：裸设备非常适合于顺序 I/O 动作。同样，OLAP 中常见的顺序 I/O(表/索引的全表扫描)使得裸设备更加适用于这种应用程序。

2) 直接读写，不需要经过操作系统级的缓存。节约了内存资源，在一定程度上避免了内存的竞争。

3) 避免了操作系统的 cache 预读功能，减少了 I/O。

4) 采用裸设备避免了文件系统的开销。比如维护 inode、空闲块等。

5) 支持并发访问，在需要多个线程或进程同时并发地访问同样的数据时，可以采用裸设备，因为裸设备没有文件系统中的 inode 锁等相关的锁概念。

2. 裸设备的缺点

- 裸设备的空间管理不灵活。在放置裸设备的时候，需要预先规划好裸设备上的空间使用。还应当保留一部分裸设备以应付突发情况，这也是对空间的浪费。
- 很多备份软件对裸设备的支持不足，导致备份等操作和方法比较原始、麻烦。

3. 文件系统的优点

文件系统易于管理和维护，可以非常方便地进行扩展，很容易实现系统的高可用和采用存储技术进行灾备。目前主要版本的 Linux 和 AIX 操作系统中默认就提供了 CIO 文件系统的访问方式，这已经和裸设备的性能不相上下，所以建议大家尽量采用文件系统作为 DB2 表空间的容器。

2.2.5 负载均衡及条带化(Striping)

当多个进程或线程同时访问磁盘时，会出现磁盘冲突。大多数磁盘系统都对访问次数(每秒的 I/O 操作)和数据传输率(每秒传输的数据量)有限制。当达到这些限制时，后面需要访问磁盘的进程就需要等待，这就是所谓的磁盘冲突。

避免磁盘冲突是优化 I/O 性能的重要手段，为了最大程度地避免磁盘冲突，就需要将所有的 I/O 访问负载均衡分担到所有可用的磁盘上，也就是 I/O 负载均衡。在一些成熟的磁盘负载均衡技术出现之前，DBA 需要了解并预测各系统的 I/O 负载量，通过手工配置每个数据到不同存放位置以分担 I/O 负载，进而达到负载均衡的目的。

而目前实现这一需求的最普遍技术就是条带化技术，也就是将一块连续的数据分成很多小部分，并把它们分别存储到不同磁盘上。这样进程在访问数据的时候，可以同时向多个不同部分同时发出 I/O 请求，由于这些数据存放在不同的磁盘上，因而不会造成磁盘冲突，而且在需要对这种数据进行顺序访问的时候可以获得最大程度上的 I/O 并行能力，从而获得非常好的性能。很多操作系统、磁盘设备供应商、各种第三方软件都能做到条带化。通过条带化，DBA 可以很轻松地做到 I/O 负载均衡而无须手工配置。

由于条带化在 I/O 性能问题上的优越表现，以至于应用系统计算环境中的多个层次或平台都涉及该技术。例如，操作系统和存储系统这两个层次中都可能使用条带化技术，图 2-22 展示了这两个层次的 I/O 结构。

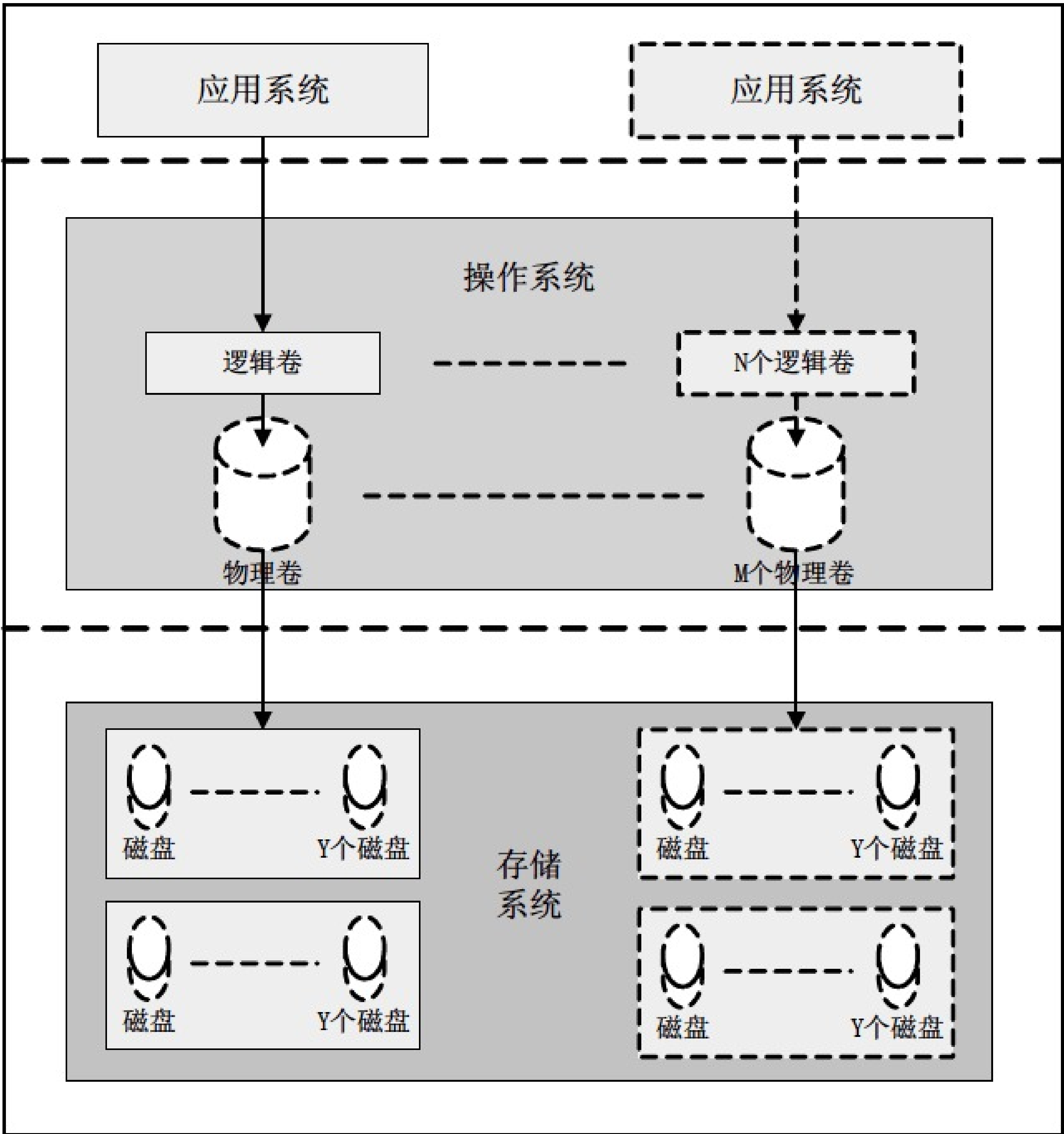


图 2-22　条带化技术在操作系统和存储系统中的应用

在介绍了存储底层的条带化之后，我们将讲解操作系统的条带化。

1. 条带化的设置

由于现在的存储技术成熟、成本降低，大多数系统都采用条带化来实现系统的 I/O 负载均衡。如果操作系统有 LVM(Logical Volume Manager，逻辑卷管理器)软件或硬件条带设备，我们就可以利用这些工具来分布 I/O 负载。当使用 LVM 或硬件条带时，影响条带的主要设置是条带深度(stripe depth)和条带宽度(stripe width)：

- 条带深度指的是条带的大小，也叫条带单元。
- 条带宽度指的是同时可以并发读或写的条带数量。这个数量等于条带集中的物理硬盘数量。

您需要根据系统的 I/O 要求来合理地选择这些数据。对于数据库系统来说，比较合理的条带深度是从 64KB 到 1MB。下面分析条带深度和条带宽度的影响因素。

2. 条带深度

为了提高 I/O 效率，我们要尽量使一次逻辑 I/O 请求转换成物理 I/O 请求后，对于一块磁盘只产生一次物理 I/O 请求。因而影响条带的重要因素就是一次逻辑 I/O 请求的大小。

此外，系统中 I/O 的并发度不同，我们对条带的配置要求也不同。例如，在高并发度且 I/O 请求的大小都比较小的情况下，我们希望一块磁盘能同时响应多个 I/O 操作；而在那些存在大 I/O 请求的低并发度系统中，我们可能就需要多块磁盘同时响应一个 I/O 请求。无论是一个磁盘还是多个磁盘响应 I/O 请求，我们的原则之一是让一次逻辑 I/O 能被一次处理完成。

为了实现这一目标，通常需要根据应用环境的具体情况进行仔细规划。

3. 条带宽度

正如我们前面所述，无论是一个还是多个磁盘响应逻辑 I/O，我们都要求 I/O 能被一次处理。因而在确定了条带深度的基础上，我们需要保证：

```
条带宽度 >= I/O 请求的大小/条带深度
```

4. 数据库对条带化的影响

下面先看以下影响 I/O 性能的操作系统和 DB2 数据库的相关参数：

- pagesize：DB2 中的数据页大小，也决定了 DB2 一次单个 I/O 请求中数据块的大小。
- prefetchsize：在预取读时，一次读取数据块的数量不能大于操作系统的最大 I/O 大小。
- sortheap：内存中 sort 区域的大小，也决定了并发排序操作时的 I/O 大小。

其中，前面两个是最关键的两个参数。

在 OLTP 系统中，会存在大量小的并发的 I/O 请求。这时就需要考虑选择比较大的条带深度。使条带深度大于 I/O 大小就称为粗粒度条带(Coarse Grain Striping)。在高并行度系统中，条带深度为 n * page size，其中 n 为大于 1 的整数。

通过粗粒度条带能实现最大的 I/O 吞吐量(一次物理 I/O 可以同时响应多个并发的逻辑 I/O)。大的条带深度能够使像全表扫描那样的预取读操作由一个磁盘驱动来响应，并提高预取读操作的性能。

在低并发度的 OLAP 系统中，由于 I/O 请求通常序列化，为了避免出现 I/O 集中的热点磁盘，我们需要避免逻辑 I/O 只由一块磁盘处理的情况发生。这时粗粒度条带就不适合了。我们选择小的条带深度，使一个逻辑 I/O 分布到尽可能多的磁盘上，从而实现 I/O 的负载均衡。这就叫细粒度条带。条带深度的大小必须为预存取参数大小(prefetchsize)的整数倍。

如果您的条带宽度设置得比较小，就需要估算出各个数据库表空间容器的 I/O 负载，并根据负载量不同将它们分别部署到不同卷上来分担 I/O 负载。

5. 操作系统对条带化的影响

- 操作系统最大 I/O 大小：决定了一次单个 I/O 请求的大小上限，不同的操作系统有不同的参数，AIX 系统是卷组的 LTG(Logical Track Group)参数。
- 操作系统级别条带化的设置。

操作系统最大 I/O 大小对条带化深度的设置也产生着重要的影响。当逻辑 I/O 请求到达操作系统之后，如果逻辑 I/O 的大小超过了操作系统所能处理的最大 I/O 大小，操作系统会根据自己能够处理的最大 I/O 大小来分割逻辑 I/O 请求。在 AIX 中，这个参数就是 LTG。也就是说，当逻辑 I/O 的大小超过了所在卷组的 LTG 设置时，AIX 会在将逻辑 I/O 以 LTG 为单位进行分割之后，才将分割好的 I/O 请求分发到物理存储中去。所以 LTG 的大小应该与条带深度大小相同或是条带深度大小的整数倍。

在 AIX V5.3 以后，卷组的 LTG 参数是在 varyonvg 的时候指定的，语法如下：

```
varyonvg -M 512K vgname
```

上面的语句将卷组“vgname”的 LTG 参数设置为 512KB，每次在 varyonvg 卷组的时候可以设置不同的 LTG(在 AIX V5.3 之前，LTG 是在创建卷组的时候设置，并且不能修改)。目前 LTG 的取值范围为：128KB、256KB、512KB、1MB、2MB、4MB、8MB、16MB、32MB 和 128MB。而且如果卷组中 PV 不能支持 LTG 的设置，varyonvg 命令会失败。

很多操作系统支持操作系统级别的条带化(而不是存储系统提供的条带化)，条带技术原理与之前介绍的无异。操作系统级别的条带化又分为逻辑条带化和物理条带化两种。逻辑条带化通常是通过 LVM 在 LV 级别实现的，而物理条带化通常是通过操作系统机器上的硬件实现的。在这里介绍一下 AIX 操作系统中 LVM 实现的逻辑条带化技术。

AIX 系统中 LVM 的基本结构如图 2-23 所示。

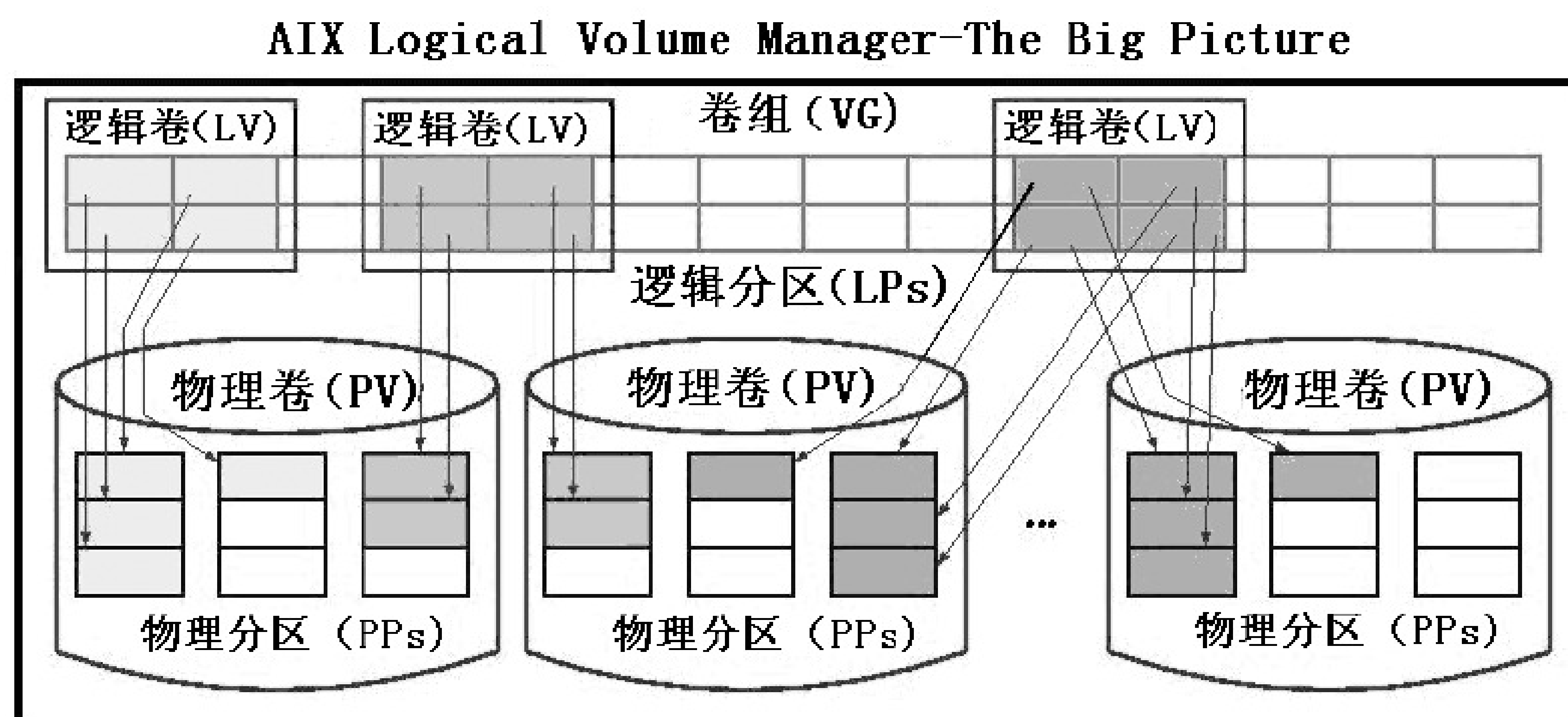


图 2-23　AIX 中 LVM 的基本结构

首先在 AIX 系统上创建 PV，再由一个或多个 PV 创建 VG，在 VG 之上再创建 LV，所以 LV 中的逻辑存储最终会映射到 PV 上的物理存储中。在不使用条带技术的情况下，LV 上连续的数据块会以图 2-24 所示的方式分布在 PV 之上。

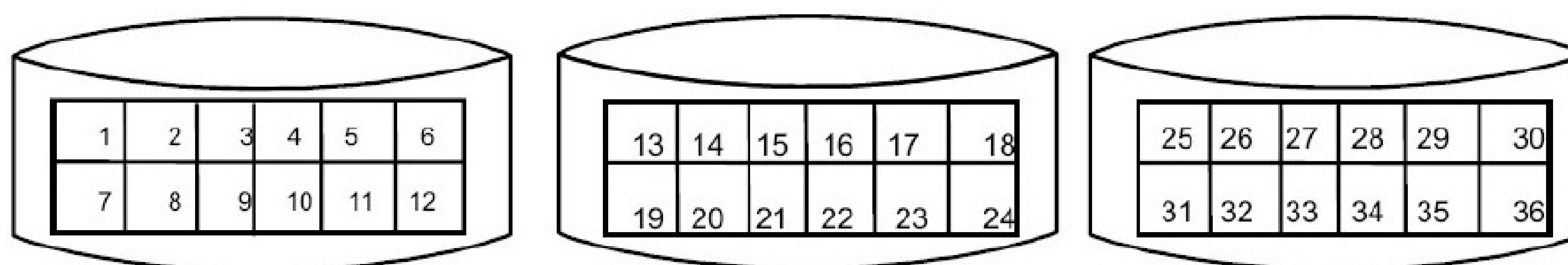


图 2-24　不使用条带技术的情况

在使用条带技术之后，LV 上连续的数据块就会以图 2-25 所示的方式分散地分布在这些 PV 之上。

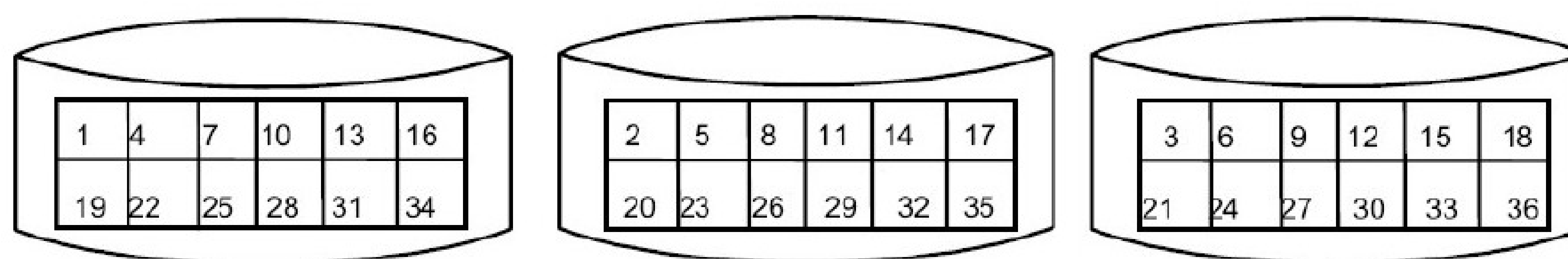


图 2-25　使用条带技术之后的情况

这样在对 LV 上连续的数据进行读或写的时候，就可以实现同时在多个 PV 上并行地处理 I/O 请求了，从而可以极大地提高 I/O 的性能。

上面已经提到的关于条带深度和条带宽度的特点在 LV 上同样适用。只是条带化处理的 I/O 不是发送到物理磁盘上，而是发送到 PV 上。在操作系统最简单的存储方案中，PV 对应的就是物理磁盘，而现在普遍使用了单独的存储系统之后，PV 通常只是逻辑磁盘 LUN，而这个逻辑磁盘 LUN 通常又是以某种条带化技术映射到了存储系统中的多个物理

磁盘上。所以在操作系统中使用了 LV 级别的条带化之后，一定要保证操作系统中条带的设置与存储系统中条带的设置相匹配，否则性能将无法得到保证。

那么我们是否需要在使用了存储系统条带技术的同时再使用操作系统级别的条带技术呢？答案是：不一定。条带化技术的目的就是最大限度地将 I/O 负载均衡，实现最大化的 I/O 并行处理。只要有利于这个目标，我们就可以同时进行这两个不同层次的条带化。相反，如果在同时使用了这两个层次的条带化后并没有实现更好的 I/O 性能，那么就没有必要了，因为这进一步增加了 I/O 结构的复杂性，更不利于存储空间的管理。

在使用了单独的存储系统之后，操作系统级别的条带化带给我们最大的性能优势就是 LV 上的 I/O 可以在多个 PV 上并行执行，而每个 PV 又映射到不同的一组物理磁盘上，这样就可以将 LV 上的 I/O 分布在更多的物理磁盘上，这对提高 I/O 的性能有极大帮助。如果 LV 分布的多个 PV 都映射到同一组的物理磁盘上，那么对 I/O 性能的提高帮助不大。

在使用了操作系统级别的条带化之后，还需要重点考虑的就是条带深度，操作系统级别条带深度的大小决定了向 PV 发送 I/O 的大小。为了获得最好的性能，就要进一步地分析 PV 在映射到存储系统时条带深度的设置，最好的结果就是：操作系统条带深度=存储系统条带深度×存储系统条带宽度。这样就能最大限度地保证两个层次条带深度的匹配，从而有最好的 I/O 性能。

在操作系统中设置条带化时，需要注意下面的原则：

- 操作系统的条带块的大小设置大些，应该是硬件条带块大小的整数倍。
- 必须和应用的特点相结合(例如，应用是随机读还是连续读，读的块的大小等)，条带技术对大量的顺序读/写有最好的 I/O 性能。

条带化技术到目前为止依然是提高 I/O 性能最好的一种技术，它通过最大化 I/O 并行特性发挥了所有可以利用的硬件的能力，所以条带化技术在各种环境中都得到了广泛的应用。我们在考虑性能问题时不可错过对条带化技术的关注。

2.3　逻辑卷和 lvmo 优化

逻辑卷层位于应用程序层和物理层之间。在磁盘 I/O 的上下文中，应用程序层是文件系统或原始逻辑卷。物理层由实际的磁盘组成。LVM 是一种 AIX 磁盘管理系统，可以在逻辑和物理存储之间映射数据。这允许数据保存在多个物理盘片上，并使用专门的 LVM 命令对其进行管理和分析。实际上，LVM 控制系统中所有的物理磁盘资源，并帮助提供存储子系统的逻辑视图。

了解了逻辑卷层位于应用程序层和物理层之间，应该可以帮助您理解它为什么很可能是所有层中最重要的一层。可以说您的物理卷本身就是逻辑层的一部分，因为物理层仅包

含实际的磁盘、设备驱动程序和任何您可能已经配置的阵列。图 2-26 阐释了这个概念，并显示了逻辑 I/O 组件与物理磁盘及其应用程序层是如何紧密结合在一起的。

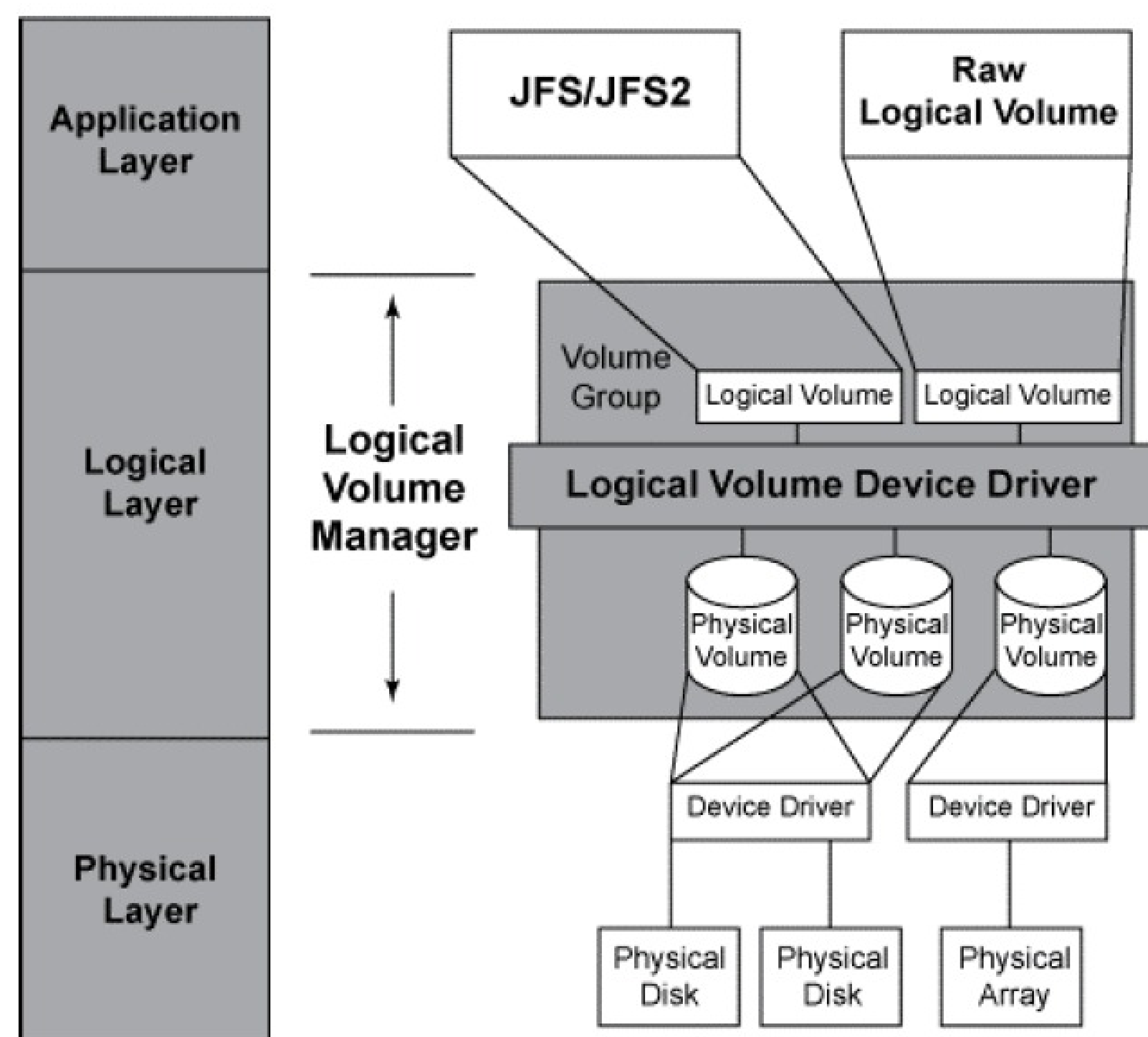


图 2-26　逻辑卷示意图

现在，让我们简要地、自底向上地介绍 LVM 中的各个元素。每个驱动器作为物理卷进行命名。多个物理卷组成卷组。在卷组中，定义了逻辑卷。LVM 允许数据位于多个物理驱动器上，尽管可能将它们配置为属于单个卷组。这些逻辑卷可以是一个或多个逻辑分区。每个逻辑分区具有与其相关联的物理分区。在其中，您可以拥有物理部分的多个副本以用于各种目的，如磁盘镜像。

2.3.1　使用 lvmo 进行优化

lvmo 是 AIX Version 5.3 中首次引入的新命令之一。lvmo 用于设置和显示您的 pbuf 优化参数，它还可以用于阻塞 I/O 统计信息。请务必注意，使用 lvmo 命令只允许更改那些专门用于特定卷组的 LVM pbuf 可调参数，ioo 实用工具仍然是在系统范围内管理 pbuf 的唯一方法。这是因为，在 AIX Version 5.3 之前，pbuf 池参数是一种系统范围的资源。随着 AIX Version 5.3 的出现，LVM 可以为每个卷组管理 pbuf 池。什么是 pbuf？最准确地说，pbuf 是固定的内存缓冲区。LVM 使用这些 pbuf 来控制挂起的磁盘 I/O 操作。

让我们显示一下 data2vg 卷组的 lvmo 可调参数：

```
# lvmo -v data2vg -a
```

```
vgname = data2vg
pv_pbuf_count  = 1024
total vg pbubs = 1024
max_vg_pbuf_count = 8192
perv_blocked_io_count = 7455
global pbuf count = 1024
global_blocked_io_count = 7455
```

其中哪些是可调参数?

- pv_pbuf_count：报告在将物理卷添加到卷组时添加的 pbuf 数目。
- max_vg_pbuf_count：报告可以为卷组分配的最大 pbuf 量。
- global_pbuf_count：报告在将物理卷添加到任何卷组时添加的 pbuf 数目。

2.3.2　卷组 pbuf 池

通常，LVM 为每个单独的 I/O 请求使用一个 pbuf，而不管传输的数据量有多大。在向 VG 添加新的 PV 时，AIX 创建了额外的 pbuf。在以前的 AIX 发布版中，pbuf 池是一种系统范围的资源，但是在 AIX 5L Version 5.3 中，LVM 为每个 VG 分配和管理一个 pbuf 池。对于那些具有大量 VG 的系统来说，这种增强可以支持高级的可扩展性和性能，并且适用于所有的 VG 类型。因为有了新的 pbuf 池的实现，AIX 可以显示和管理附加的 LVM 统计信息和调整参数。

lvmo 命令为与新的 pbuf 池相关的管理任务提供了支持。您可以使用 lvmo 命令来显示 pbuf 和阻塞 I/O 的统计信息，以及 pbuf 可调参数的设置，而不管该实体的范围是系统范围还是 VG 特定的。然而，lvmo 命令只允许更改专门用于特定 VG 的 LVM pbuf 可调参数的设置。ioo 命令继续管理系统范围的单独的 pbuf 可调参数。另外，与以前的 AIX 发布版本一样，vmstat -v 命令仍用于显示整个系统中由于缺少空闲 pbuf 而阻塞的 I/O 的数目。

让我们为下面这个卷组增加 pbuf 计数：

```
# lvmo -v redvg -o pv pbuf count=2048
# vmstat -v
            6029312 memory pages
            5734766 lruable pages
            2801540 free pages
                  4 memory pools
             406918 pinned pages
               80.0 maxpin percentage
```

```
                     20.0 minperm percentage
                     80.0 maxperm percentage
                      2.3 numperm percentage
                   135417 file pages
                      0.0 compressed percentage
                        0 compressed pages
                      0.0 numclient percentage
                     80.0 maxclient percentage
                        0 client pages
                        0 remote pageouts scheduled
                   312417 pending disk I/Os blocked with no pbuf
                        0 paging space I/Os blocked with no psbuf
                     2878 filesystem I/Os blocked with no fsbuf
                        0 client filesystem I/Os blocked with no fsbuf
                        0 external pager filesystem I/Os blocked with no fsbuf
```

2.3.3 pbuf 设置不合理导致性能问题调整案例

下面我们举一个实际的案例。

客户应用系统环境：

- 存储：日立存储 HDS
- 操作系统：IBM AIX
- 数据库：DB2 V8.1.9
- 多路径通道软件：HDLM

提示：

HDLM(Hitachi Dynamic Link Manager)是 HDS 公司提供的安装在主机端的存储通道传输管理工具软件。HDLM 提供主机到存储系统的 I/O 通道负载平衡和故障切换功能，增强了主机系统的数据可得性。

<table>
<tr><td>问题描述：
2007/09/23：8 点 10 分左右，CSR SAPR3 节点，DB2 数据库出现挂起，同时操作系统有大量 LVDD 报错。</td></tr>
<tr><td>问题诊断：
(1) 从 error log 看文件系统损坏之前，发生了什么事件。
D2A1B43E 0623084108 P U SYSPFS FILE SYSTEM CORRUPTION
EA88F829 0623081708 I O SYSJ2 USER DATAI/OERROR
D2A1B43E 0623081708 P U SYSPFS FILE SYSTEM CORRUPTION</td></tr>
</table>

```
613E5F38     0623081708 N U LVDD
A39F8A49     0623081708 T S syserrlg     ERROR LOGGING BUFFER OVERFLOW
613E5F38     0623081708 N U LVDD
 ......          <<<           大量的LVDD 错误
613E5F38     0623081708 N U LVDD
613E5F38     0623081708 N U LVDD
 .....           <<<           大量的LVDD 错误
613E5F38     0623081708 N U LVDD
613E5F38     0623081708 N U LVDD
 ..........      <<<           大量的LVDD 错误
613E5F38     0623081708 N U LVDD
613E5F38     0623081708 N U LVDD
 ...........     <<<           大量的LVDD 错误
613E5F38     0623081708 N U LVDD
0EC00096     0623081708 P U SYSPFS     STORAGE SUBSYSTEM FAILURE
<<< Jun 23 08:17:31 JFS_META_EXCEPTION
A39F8A49     0623081708 T S syserrlg     ERROR LOGGING BUFFER OVERFLOW
52715FA5     0623081708 U H LVDD      FAILED TO WRITE VOLUME GROUP
STATUS AREA   <<< Jun 23 08:17:30 更新VGSA 失败
613E5F38     0623081708 N U LVDD
613E5F38     0623081708 N U LVDD
 .......         <<<           大量的LVDD 错误
613E5F38     0623081708 N U LVDD
613E5F38     0623081708 N U LVDD
 ......          <<<           大量的LVDD 错误
613E5F38     0623081708 N U LVDD
 .....           <<<           大量的LVDD 错误
613E5F38     0623081708 N U LVDD
613E5F38     0623081708 N U LVDD     << Sep 23 08:17:39 LVDD 错误
3074FEB7     0623081608 T H fscsi0          ADAPTER ERROR
3074FEB7     0623080908 T H fscsi1          ADAPTER ERROR
<< Sep 23 08:09:38 FSCSI_ERR4 错误
B8FBD189     0623080908 T S fscsi1          SOFTWARE PROGRAM ERROR
```

<< Sep 23 08:09:38 FSCSI_ERR6 错误

A63BEB70 0621011608 N U SYSPROC

(2) 文件系统损坏之前发生的事件过程如下：

08:09:38 HBA卡到光纤交换机的通信临时中断，发生路径切换。

08:17:30 产生LVDD 错误，AIX LVM 的I/O 请求无法完成。

指向的设备是： dlmfdrv109 00c1ef4b81ac62f2 ocrvg11

08:17:30 更新VGSA 失败

指向的设备是：

brw-r--r-- 1 root system 42,181 Jun 16 01:59 dlmfdrv180

dlmfdrv180 00c19d3b542ae30b ocrvg01 active

08:17:31 JFS_META_EXCEPTION，LVM 在读取JFS 文件系统的meta data时产生I/O错误。

#define EIO 5 /*I/Oerror */

(3) 分析具体原因。

HBA 卡到光纤交换机通信中断，error log记录的原因是4E。

Errno 4E = ETIMEDOUT

原因如下：

AIX LVM 发送给HDLM的I/O请求，因为HDLM 的参数hd_pbuf_cn到达了上限而失败。如果hd_pbuf_cn到达了上限，我们就会在error log中看到大量的lvdd Error NONE (613E5F38)。hd_pbuf_cn 这个参数的合理值为65536，在AIX 5.3中，与hd_pbuf_cn 相关的参数是max_vg_pbuf_count和hd_pbuf_cnt。

hd_pbuf_cnt和在AIX 5L Version 5.3 中的max_vg_pbuf_count有如下限制：并发I/O请求的最大数量的允许值是从16384 到1000000。默认值是16384。

9月23日，SAPR3 的HDLM 的pbuf count设置值如下：

```
root@SAPR3:/> /usr/DynamicLinkManager/bin/dlmodmset -o
Inquiry Log              : on
Inquiry Log File Size    : 1000
hdisk error check flag   : off
HDLM pbuf count          : 16384
Lun Reset                : off
KAPL10800-I The dlmodmset utility completed normally.
root@SAPR3:/>
```

诊断结论：

在问题出现前，CSR系统做了系统变更(通过LVM mirror做了数据保护)和业务操作(正在做数据库恢复)，这些都增加了并发写I/O的量。我们得到这样的分析结果：业务的变化

使CSR系统SAPR3节点的并发I/O数量大大增加，达到了HDLM参数hd_pbuf_cn设置的上限16384，导致LVM送达HDLM的I/O失败，包括更新VGSA 及文件系统meta的I/O操作也同时失败，造成卷组中stale pp的产生和文件系统的损坏。

调整建议：

- 增大 HDLM pbuf count 的值为 65536。
- 同时相应增大操作系统与之对应的 max_vg_pbuf_count 参数值到 65536。

调整步骤：

方法：

(1) 增大VG的pbufs

(2) max_vg_pbuf_count - max pbufs available for the VGs, requires varyon/varyoff

lvmo -v <vgname> -o max_vg_pbuf_count=<new value>

步骤：需要对如下vg进行调整

ocrvg02

ocrvg01

(3) lvmo -v ocrvg01 -o max_vg_pbuf_count=65536

(4) varyoffvg ocrvg01

(5) varyonvg ocrvg01

(6) lvmo -v ocrvg02 -o max_vg_pbuf_count=65536

(7) varyoffvg ocrvg02

(8) varyonvg ocrvg02

为使更新生效，需要重新配置 HDLM driver 或重新启动系统。

调整效果：

经过上述调整，该系统的 I/O 吞吐量加大，在业务相同的情况下连续观察几天没有再出现类似的错误。同时性能也大大提高。

从上面这个案例中，我们可以看到，造成系统出现错误的主要原因是操作系统层面卷组的 pbuf 和存储层面 hd_puf_cn 的“边界”配合出现问题。所以希望大家通过这个案例能对存储层面和操作系统的性能调整有更深入的了解。

2.3.4　使用 ioo 进行优化

表 2-1 列出了一些特定的优化参数，通过调整这些参数可以提高 I/O 性能。

表 2-1　特定的优化参数

功　　能	JFS 优化参数	增强的 JFS2 优化参数
设置缓存文件的最大内存容量	vmo -o maxperm=value	vmo -o maxclient=value(小于或等于 maxperm)
设置缓存文件的最小内存容量	vmo -o minperm=value	不适用
设置缓存的内存限制(硬限制)	vmo -o strict_maxperm	vmo -o maxclient(硬限制)
设置用于提前顺序读取的最大页面数	ioo -o maxpgahead=value	ioo -o j2_maxPageReadAhead=value
设置用于提前顺序读取的最小页面数	ioo -o minpgahead	ioo -o j2_minPageReadAhead=value
设置对于文件的挂起写 I/O 的最大数目	chhdev -l sys0 -a maxpout maxpout	chdev -l sys0 -a maxpout maxpout
设置对于文件的挂起写 I/O 的最小数目，在此情况下，由 maxpout 阻塞的程序可以继续	chdev -l sys0 -a minpout minpout	chdev -l sys0 -a minpout minpout
使用随机写操作为文件设置修改数据缓存的容量	ioo -o maxrandwrt=value	ioo -o j2_maxRandomWrite ioo -o j2_nRandomCluster
为延迟的顺序写操作控制 I/O 的收集	ioo -o numclust=value	ioo -o j2_nPagesPerWriteBehindCluster=value
设置 f/s bufstruct 的数目	ioo -o numfsbufs=value	ioo -o j2_nBufferPerPagerDevice=value

您可以使用几种不同的方式来确定系统中现有的 ioo 值。ioo 的显示清单很长，其中清晰地提供了大部分信息。它列出了当前值、重新启动值、范围、单位、类型和由 ioo 管理的所有可调整参数的依赖关系。

```
> ioo -L
NAME       CUR    DEF    BOOT   MIN    MAX    UNIT          TYPE    DEPENDENCIES
j2 atimeUpdateSymlink    0    0     0     0     1     boolean          D
j2 dynamicBufferPreallo  16   16    16    0     256   16K slabs        D
j2 inodeCacheSize        400  400   400   1     1000                   D
j2_maxPageReadAhead      128  128   128   0     64K   4KB pages        D
```

```
j2 maxRandomWrite         0    0     0    0    64K    4KB pages      D
j2 maxUsableMaxTransfer   512  512   512  1    4K     pages          M
j2 metadataCacheSize      400  400   400  1    1000                  D
j2 minPageReadAhead       2    2     2    0    64K    4KB pages      D
j2 nBufferPerPagerDevice 512  512   512  512  256K                  M
j2 nPagesPerWriteBehindC 32   32    32   0    64K                   D
j2 nRandomCluster         0    0     0    0    64K    16KB clusters  D
j2 nonFatalCrashesSystem 0    0     0    0    1      boolean        D
j2 syncModifiedMapped     1    1     1    0    1      boolean        D
j2 syncdLogSyncInterval   1    1     1    0    4K     iterations     D
jfs clread enabled        0    0     0    0    1      boolean        D
jfs use read lock         1    1     1    0    1      boolean        D
lvm bufcnt                9    9     9    1    64     128KB/buffer   D
maxpgahead minpgahead     8    8     8    0    4K     4KB pages      D
maxrandwrt                0    0     0    0    512K   4KB pages      D
memory frames             512K       512K             4KB pages      S
Minpgahead maxpgahead     2    2     2    0    4K     4KB pages      D
numclust                  1    1     1    0    2G-1   16KB/cluster   D
numfsbufs                 196  196   196  1    2G-1                  M
pd npages                 64K  64K   64K  1    512K   4KB pages      D
pgahd scale thresh        0    0     0    0    419430 4KB pages      D
pv min pbuf               512  512   512  512  2G-1                  D
sync release ilock        0    0     0    0    1      boolean        D

n/a means parameter not supported by the current platform or kernel
Parameter types:
    S = Static: cannot be changed
    D = Dynamic: can be freely changed
    B = Bosboot: can only be changed using bosboot and reboot
    R = Reboot: can only be changed during reboot
    C = Connect: changes are only effective for future socket connections
    M = Mount: changes are only effective for future mountings
    I = Incremental: can only be incremented
    d = deprecated: deprecated and cannot be changed
```

下面的代码向您显示了如何更改可调整的参数：

```
ioo -o maxpgahead=32
Setting maxpgahead to 32
```

上面这个参数仅用于 JFS。接下来的部分将适用于 JFS2。

一些重要的、JFS2 特定的文件系统的性能增强功能包括，提前顺序页面读取和延迟顺序、随机写入。通过观察文件的访问模式，AIX 的虚拟内存管理器(Virtual Memory Manager，VMM)可以预测页面需求。当程序访问文件的两个页面时，VMM 假定该程序将采用顺序的方法不断地尝试访问该文件。可以使用 VMM 阈值来配置将要提前读取的页面数目。对于 JFS2，记录以下两个重要参数：

- j2_minPageReadAhead：用于确定当 VMM 最初检测到顺序模式时提前读取的页面数目。
- j2_maxPageReadAhead：用于确定 VMM 可以在顺序文件中读取页面的最大数量。

1. 文件同步性能调优

JFS 的非顺序文件 I/O 会一直存储在内存中，直到满足一定条件：

- 空闲列表缩小到 minfree，以至于需要进行页替换。
- syncd 守护程序按固定调度间隔刷新页。
- 执行了 sync 命令。
- 在达到参数 j2_maxRandomwrite 设置的阈值后，会启用内存中的脏页。

如果在以上的任一条件满足前已存储了过多页，那么在 syncd 守护程序进行刷新时，会获得 i-node 锁，并保持到所有的脏页面都被写入磁盘。在这段时间里，任何试图访问此文件的线程会由于无法获得 i-node 锁而被阻塞。请记住：syncd 守护程序会顺利地刷新文件中的所有脏页面，但限于一次一个文件。在拥有大量内存并同时有大量页需要修改的系统中，在 syncd 守护程序刷新页时，I/O 可能达到高峰值。

AIX 含有名为 sync_release_ilock 的可调选项。ioo 命令加上-o sync_release_ilock=1 选项允许在清空该文件的脏页面后释放 i-node 锁。这一选项使得在调用 sync()的过程中访问该文件有更好的响应。

阻塞效果也可通过在 syncd 守护程序中提高同步频率使之最小化。更换用于启用 syncd 守护程序的/sbin/rc.boot，然后重新引导系统使之生效。对现行系统，杀死 syncd 守护程序进程并按新的值重新启动守护程序。

2. 文件系统缓冲区调优

以下 ioo 参数可用于调优磁盘 I/O：

numfsbufs 参数

当有大量针对文件系统的同步或大型 I/O，或是存在针对文件系统的大型顺序 I/O 时，这些 I/O 可能会在等待 bufstruct 时成为文件系统级的瓶颈。每个文件系统的 bufstructs 数目(称为 numfsbufs)可使用 ioo 命令增加。该值仅在文件系统加载后才会生效，因此，如果

更改了这个值，就必须卸载，然后再次加载文件系统。numfsbufs 默认每个文件系统有 93 个 bufstruct。

j2_nBufferPerPagerDevice 参数

在增强型 JFS 中，bufstruct 的数量由参数 j2_nBufferPerPagerDevice 指定。当前增强型 JFS 文件系统的默认 bufstruct 数是 512。每个增强型 JFS 文件系统的 bufstructs 数(j2_nBufferPerPagerDevice)可以使用 ioo 命令来增加。该值在文件系统被加载后才起作用。

lvm_bufcnt 参数

如果应用程序正在处理大量的裸 I/O 而不通过文件系统，同文件系统相同类型的瓶颈也可能出现在 LVM 层上。极大量的 I/O 加上极快的 I/O 设备可能会导致 LVM 层上的瓶颈。但是如果真的出现瓶颈，那么可以通过 ioo 命令增加 lvm_bufcnt 参数来提供大量的“uphysio”缓冲区。该值会立刻生效。当前的默认值是 9 个“uphysio”缓冲区。由于当前 LVM 将 I/O 分为每个 128KB，而 lvm_bufcnt 的默认值为 9，故一次可写入 9*128KB。如果正在进行的 I/O 大于 9*128 KB，增加 lvm_bufcnt 的值才会有利。

hd_pbuf_cnt 参数

hd_pbuf_cnt 参数控制可用于 LVM 设备驱动程序的 pbufs 数。pbuf 是用于存放暂挂于 LVM 层的 I/O 请求的固定内存缓冲区。

在 AIX 中，顺序 I/O 的结合使得无论 I/O 包括多少页，每个顺序 I/O 请求只使用单个 pbuf。这种类型的瓶颈一般很难遇到。而对于随机 I/O，除非运行 syncd 守护程序，I/O 一般会被零星地刷新。

确定是否发生 pbuf 瓶颈的最好方法是检查被称为 hd_pendqblked 的 LVM 变量。以下代码会给出该变量的值：

```
#!/bin/ksh
# requires root authority to run
# determines number of times LVM had to wait on pbufs since system boot
addr='echo "knlist hd pendqblked" | /usr/sbin/crash 2>/dev/null |tail -1|
     cut -f2 -d: '
value='echo "od $addr 1 D" | /usr/sbin/crash 2>/dev/null | tail -1| cut -f2
       -d: '
echo "Number of waits on LVM pbufs are: $value"
exit 0
```

ioo -a 命令也可以显示 hd_pendqblked 值。

注意：
请不要把 hd_pbuf_cnt 的值设得太大，因为除了重新引导系统外无法减小该值。

pd_npages 参数

pd_npages 参数指定当删除文件时 RAM 的某一块中应该删除的页数。改变该值只对那些需要删除文件的实时应用程序有用。由于在分派某个进程 / 线程之前将删除少量的页面，因此通过减小 pd_npages 参数的值，实时应用程序可获得更快的响应时间。默认值是最大可能文件大小除以页面大小(目前为 4096)；如果最大可能文件大小为 2GB，那么 pd_npages 参数的值默认为 524288。

v_pinshm 参数

当 v_pinshm 参数设置为 1 时，如果执行 shmget()的应用程序指定 SHM_PIN 作为标志的一部分，就会使共享内存段中的页面由 VMM 固定。默认值为 0。

fsbufwaitcnt 和 psbufwaitcnt 计数器

只要 bufstruct 变得不可用以及 VMM 将线程放入 VMM 等待列表中，fsbufwaitcnt 和 psbufwaitcnt 计数器就会递增。使用 crash 命令或 ioo -a 命令的 fsbufwaitcnt 和 psbufwaitcnt 选项来检查这些计数器的值。下面是输出示例：

```
# ioo -a
hd pendqblked = 305
psbufwaitcnt = 0
fsbufwaitcnt = 337
xpagerbufwaitcnt 计数器
```

只要增强型 JFS 文件系统上的 bufstruct 不可用，xpagerbufwaitcnt 就会递增。可使用 ioo -a 命令检查 xpagerbufwaitcnt 计数器的值。下面是输出示例：

```
# ioo -a
xpagerbufwaitcnt = 815
```

2.4 操作系统性能调整总结

在应用系统出现性能问题时，操作系统是我们定位问题的入口，它在整个应用系统中起到承上启下的作用。我们首先要通过操作系统定位性能瓶颈，然后才能展开下一步的调

整工作。同时，在操作系统上也有很多可以优化的地方，例如 I/O 访问方式、条带化，以及相关逻辑卷配置参数等都可以通过合理调整来提高应用系统的性能。

2.5　存储 I/O 设计

对于任何程序的运行来说，最慢、最花费时间的操作实际上是从磁盘中检索数据。这主要缘于磁盘 I/O 访问中存在的物理机械过程(磁头旋转和寻道)。尽管磁盘存储技术在最近几年取得了极大的进步，但磁盘的旋转速度却没有太大的提高。您必须清楚这样一个事实：在一定条件下，RAM 访问仅仅需要大概 540 个 CPU 时钟周期，而磁盘访问则需要花费大概 20 000 000 个 CPU 时钟周期。很明显，系统中访问数据最薄弱的环节就是磁盘 I/O 存储系统，从性能调整的角度来说，就是确保磁盘数据布局不会成为更严重的瓶颈。糟糕的数据布局将会给 I/O 性能带来更大的影响。在对系统进行任何优化活动之前，首先应该了解您的存储 I/O 系统的物理体系结构，因为如果您所设计的存储 I/O 系统非常糟糕，并且其中包含慢速磁盘，或者适配器的使用非常低效，那么其他的任何优化工作都无法提供帮助。所以良好的存储 I/O 设计对于系统性能的提升是很显著的。

2.6　存储基本概念

关于存储的概念太多，许多已经超出了本书的讨论范围。本章主要讲解最常见的几个概念，它们是我们进行存储 I/O 设计所必须掌握的。

2.6.1　硬盘

硬盘处于整个存储系统的最底层，核心的业务数据通常都存放在硬盘上。我们从硬盘上读取一次 I/O 要花费的时间如下：

硬盘上一次 I/O 时间=磁盘寻道时间+磁头旋转到特定扇区时间+传输时间+延迟

图 2-27 所示的硬盘 I/O 传输图中标识了磁头在不同位置的寻道时间。衡量磁盘的 I/O 能力有如下几个指标：

- 硬盘的转速(Rotational Speed)：也就是硬盘电机主轴的转速，转速是决定硬盘内部传输率的关键因素之一，它的快慢在很大程度上影响了硬盘的速度。同时，转速的快慢也是区分硬盘档次的重要标志之一。硬盘的主轴马达带动盘片高速旋转，产生浮力使磁头飘浮在盘片上方。要将所要存取资料的扇区带到磁头下方，转速

越快，等待时间也就越短。因此，转速在很大程度上决定了硬盘的速度。目前市场上常见的硬盘转速一般有 5400rpm、7200rpm 和 15000rpm。理论上，转速越快越好，因为较高的转速可缩短硬盘的平均寻道时间和实际读写时间。但是转速越快发热量越大，不利于散热。现在的主流硬盘转速一般为 15000rpm 以上。

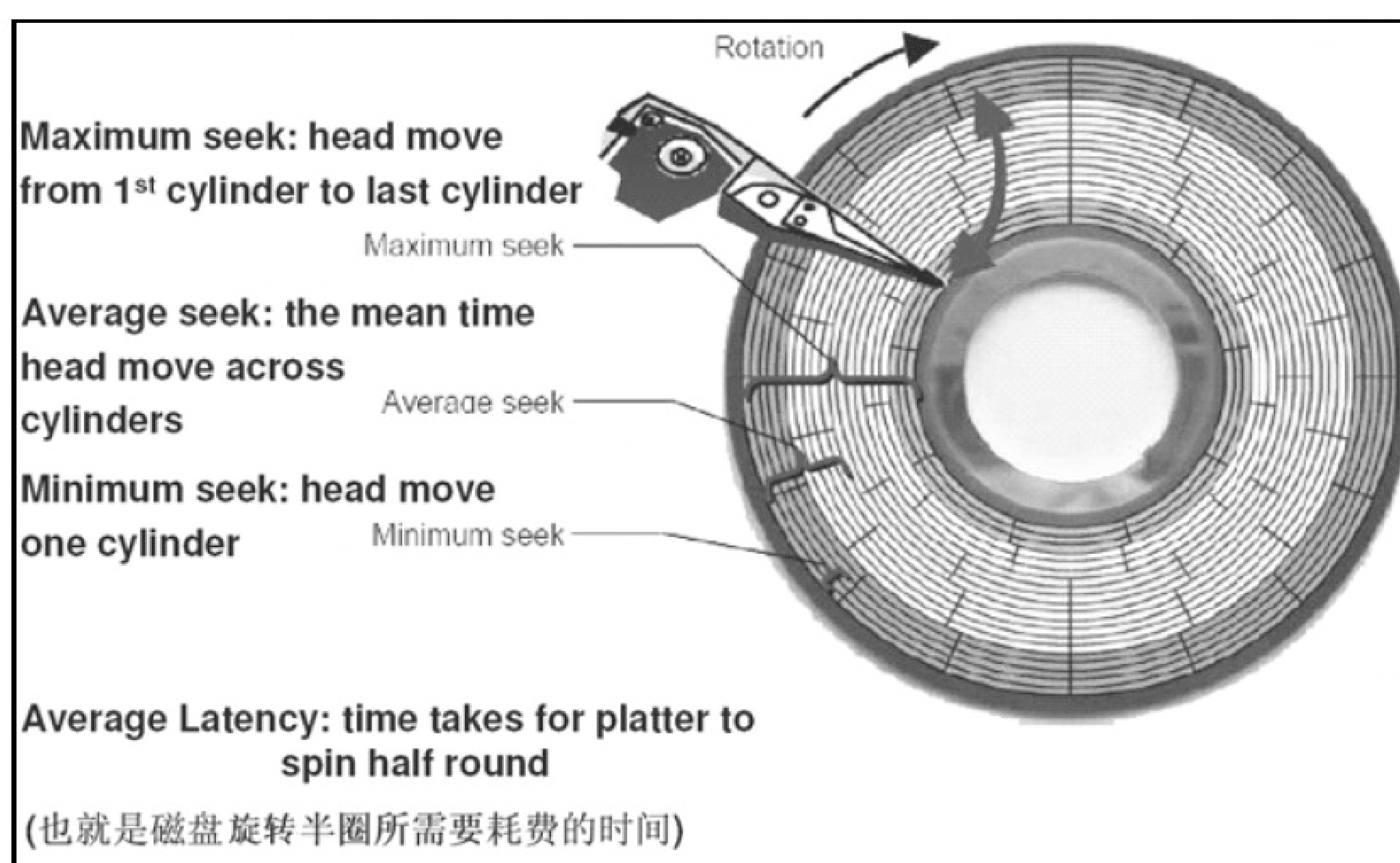


图 2-27　磁盘 I/O 传输图

- 平均寻道时间(Average Seek Time)：指硬盘在盘面上移动读写头至指定磁道寻找相应目标数据所用的时间，它描述硬盘读取数据的能力，单位为毫秒。当单碟片容量增大时，磁头的寻道动作和移动距离减少，从而使平均寻道时间减少，加快硬盘速度。
- 平均延迟时间(Average Latency Time)：指当磁头移动到数据所在的磁道后，然后等待所要的数据块继续转动到磁头下所用的时间。
- 平均访问时间(Average Access Time)：指磁头找到指定数据的平均时间，通常是平均寻道时间和平均延迟时间之和。平均访问时间最能够代表硬盘找到某一数据所用的时间，越短的平均访问时间越好。

为什么要讲硬盘呢？因为 DB2 数据库在工作时，一条 SQL 语句在经过优化器编译时，优化器会读取统计信息、数据库配置参数和相关硬件信息来为该条 SQL 语句生成最优的执行计划。其中的硬件信息包括表空间的 transrate 和 overhead。这两个参数的计算就是由硬盘的相关属性来决定的。它们的计算公式如下：

transrate=(1/传送速率)*1000/1024000*4096(假设用 4KB 页大小)

overhead=平均寻道时间+(((1/磁盘转速)*60*1000)/2)

而平均寻道时间、磁盘旋转速度和传送速率是由硬盘本身决定的，所以我们必须合理地进行存储 I/O 设计以使优化器更好地工作。

2.6.2　磁盘阵列技术

RAID 的全称是独立磁盘冗余阵列(Redundant Array of Independent Disk)。它通过将多个相对比较便宜的磁盘组合起来，并相互连接，同时都连到一个或多个计算机上，组成磁盘组，从而使性能和容量达到或超过价格更昂贵的大型磁盘。20 年来，RAID 推出了一系列级别，包括 RAID 0、RAID 1、RAID 2、RAID 3、RAID 4、RAID 5，以及各种组合(比如 RAID 1+0 等)，其中应用最广泛的是 RAID 5 和 RAID 1+0。

RAID 1+0

RAID 0 能提供更好的性能，RAID-1 能提供最佳的数据保护。如果把两者结合在一起，就能同时提供高性能和数据保护，但这也会同时提高磁盘阵列造价。

RAID 5

RAID 5 不做全磁盘镜像，但它会对每一个写操作做奇偶校验计算并写入奇偶校验数据。奇偶校验磁盘避免了像 RAID 1 那样完全重复写数据。当磁盘失效时，校验数据被用来重建数据，从而保证系统不会崩溃。为避免磁盘瓶颈，奇偶校验和数据都会被分布到阵列中的各个磁盘。尽管读的效率提高了，但是 RAID 5 需要为每个写操作做奇偶校验，因此它的写效率很差。

2.6.3　存储的 Cache

高端存储系统中除了要有高性能、高扩展性的结构外，Cache 的设计将直接影响到存储系统的性能表现和可靠性。

Cache 的主要作用是什么呢？作为缓存，Cache 的作用具体体现在读与写两个不同的方面：作为写，一般存储阵列只要求数据写到 Cache 就算完成了写操作。当写 Cache 的数据积累到一定程度，阵列才把数据刷到磁盘，这样可以实现批量写入。所以，阵列的写是非常快速的。至于 Cache 数据的保护，一般都依赖于镜像与电池(或是 UPS)。

Cache 在读数据方面的作用同样不可忽视，因为如果所要读取的数据能在 Cache 中命中的话，将大大减少磁盘寻道所需要的时间。因为磁盘从开始寻道到找到数据，一般都在

6ms 以上，而这个时间，对于那些密集型 I/O 的应用可能不是太理想。但是，如果能在 Cache 保存的数据中命中，一般响应时间则可以缩短在 1ms 以内。存储的 Cache 大小对整个 I/O 性能的影响是非常大的。

2.6.4 网络存储技术

FC SAN(Storage Area Network，存储区域网)是高速的子网，这个子网中的设备可以从您的主网卸载流量。通常，SAN 由 RAID 阵列连接光纤通道(Fibre Channel)组成，SAN 同服务器和客户机的数据通信是通过 SCSI 命令而非 TCP/IP 实现的，数据处理是“块级”(block level)。

SAN 通过特定的互联方式连接若干台存储服务器而组成单独的数据网络，提供企业级的数据存储服务，如图 2-28 所示。SAN 是一种特殊的高速网络，连接网络服务器和诸如大磁盘阵列或备份磁带库的存储设备，SAN 置于 LAN 之下，而不涉及 LAN。利用 SAN，不仅可以提供大容量的存储数据，而且地域上可以分散，缓解了大量数据传输对局域网的影响。SAN 的结构允许任何服务器连接到任何存储阵列，不管数据存放在哪里，服务器都可直接存取所需的数据。

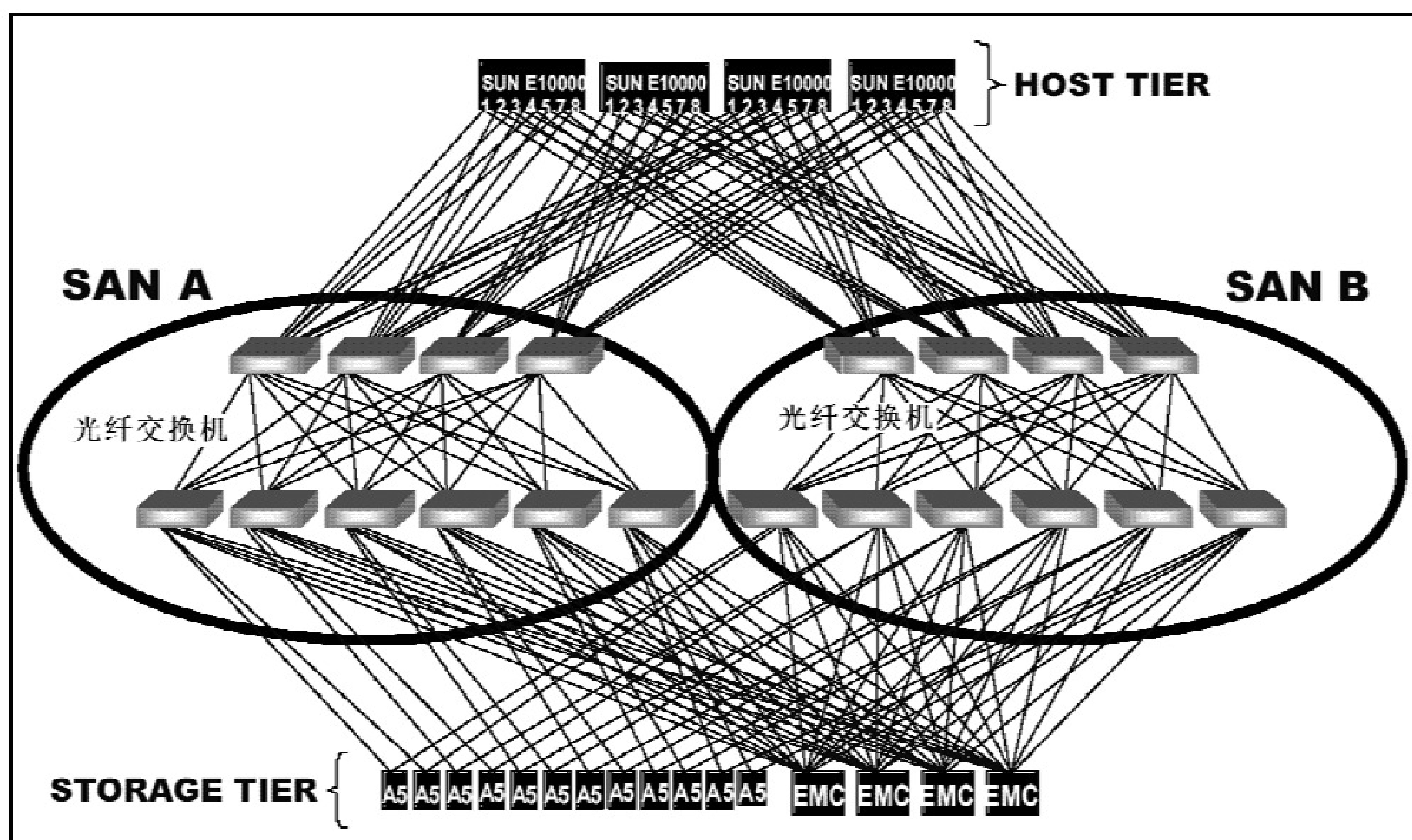


图 2-28 SAN 存储示意图

NAS 是 Network Attached Storage(网络附加存储)的简称。在 NAS 存储结构中，存储系

统不再通过 I/O 总线附属于某个服务器或客户机，而是直接通过网络接口与网络直接相连，由用户通过网络访问。NAS 是连接到计算机网络的文件层的数据存储，可以为不同网络的客户端提供数据存储服务。NAS 的硬件与传统的专用文件服务器相似，它们的不同点在于软件端。NAS 中的操作系统和其他软件只提供数据存储、数据访问功能，以及对这些功能的管理。与传统以服务器为中心的存储系统相比，数据不再通过服务器内存转发，而是直接在客户机和存储设备间传送，服务器仅起控制管理的作用。

SAN 和 NAS 的对比如表 2-2 所示。

表 2-2　SAN 和 NAS 的对比

对 比 因 素	SAN	NAS
连接介质	SCSI 或光纤	网络线缆
传输协议	FCP(光纤通道协议)	TCP/IP
数据传输类型	SCSI 数据块	文件级
数据传输速率	高	较低
集群应用支持	好	差
数据库支持	好	差

2.7　存储架构

2.7.1　存储 I/O 处理过程

了解典型的 I/O 流动图对于我们定位 I/O 性能瓶颈是非常重要的。下面我们来看下典型的 I/O 流动图，如图 2-29 所示。

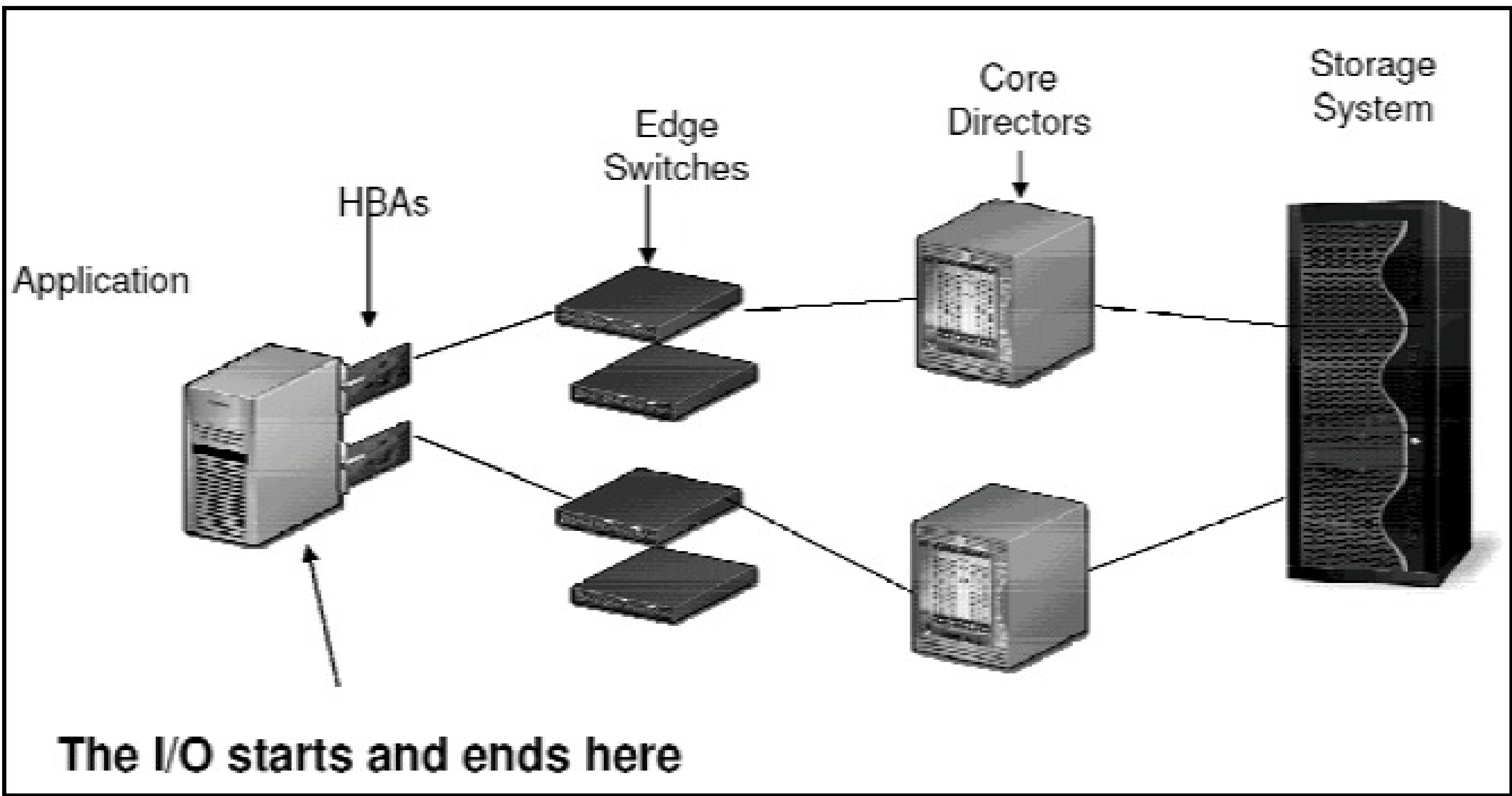


图 2-29　典型的 I/O 流动图

提示：

Core Directors 可以智能化地自动实现负载平衡和流量控制。在 I/O 处理过程中，要保证 HBA 卡的网络带宽、光交换机端口带宽、存储的前端端口、后端端口的速度匹配。不要因为某个瓶颈而导致“木桶效应”。

2.7.2 RAID IOPS

IOPS 即 I/O Per Second，也就是磁盘每秒进行读写(I/O)操作的次数，决定 IOPS 的主要因素在于阵列的算法、Cache 命中率以及磁盘个数。阵列的算法会由于阵列的不同而不同，例如在 hds usp 上面，可能因为 ldev(lun)存在队列或资源限制，而导致单个 ldev 的 IOPS 上不去。所以我们在进行存储 I/O 设计时，必须结合我们使用的存储产品去了解这个存储的一些算法规则与限制。

Cache 的命中率取决于数据的分布、Cache 的大小、数据访问的规则以及 Cache 的算法。如果完整讨论下来，这里将变得很复杂，超出了本书的讨论范围。我这里只强调 Cache 的命中率，对于阵列，读 Cache 的命中率越高越好，这一般表示它可以支持更多的 IOPS，为什么这么说呢？这就与我们下面要讨论的硬盘 IOPS 有关系了。

硬盘的限制，每个物理硬盘能处理的 IOPS 是有限制的，比如：

```
10Krpm        15Krpm        ATA
10M/s         13M/s         8M/s
10Krpm        15Krpm        ATA
100IOPS       150IOPS       50IOPS
```

同样，如果一个阵列有 120 块 15K rpm 的光纤硬盘，那么这个陈列能支撑的最大 IOPS 为 120*150=18000，这是硬件限制的理论值，如果超过这个值，硬盘的响应可能会变得非常缓慢而不能正常提供业务。

在 RAID 5 与 RAID 10 上，读 IOPS 没有差别。但是，相同的业务写 IOPS，最终落在磁盘上的 IOPS 却是有差别的，而我们评估的正是磁盘的 IOPS，如果达到了磁盘的限制，性能肯定是上不去了。

我们在第 1 章中手机银行系统的 IOPS 是 10000，读 Cache 的命中率是 30%，读 IOPS 为 60%，写 IOPS 为 40%，磁盘个数为 120。我们分别计算在 RAID 5 与 RAID 10 的情况下，每个磁盘的 IOPS 为多少。

RAID 5

单块盘的 IOPS = (10000*(1 – 0.3)*0.6 + 4 * (10000*0.4))/120

= (4200 + 16000)/120

= 168

这里的 10000*(1－0.3)*0.6 表示是读的 IOPS，比例是 0.6，除掉 Cache 命中，实际只有 4200 个 IOPS。而 4 * (10000*0.4) 表示写的 IOPS，因为每一个写，在 RAID 5 中，实际发生了 4 个 IO，所以写的 IOPS 为 16000 个。为了考虑 RAID 5 在写操作的时候，那两个读操作也可能发生命中，所以更精确的计算为：

单块盘的 IOPS= (10000*(1－0.3)*0.6 + 2 * (10000*0.4)*(1－0.3) + 2 * (10000*0.4))/120

= (4200 + 5600 + 8000)/120

= 148

计算出来单个盘的 IOPS 为 148 个，基本达到磁盘极限。

RAID 10

单块盘的 IOPS = (10000*(1-0.3)*0.6 + 2 * (10000*0.4))/120

= (4200 + 8000)/120

= 102

可以看到，因为 RAID 10 对于写操作。只发生两次 IO，所以同样的压力，同样的磁盘，每个盘的 IOPS 只有 102 个，还远远低于磁盘的极限 IOPS。

2.7.3　RAID 10 和 RAID 5 的比较

在我们考虑存储 I/O 设计时，到底用 RAID 10 还是选择 RAID 5 呢？下面我们来做个比较。

为了方便对比，这里拿同样多驱动器的磁盘来做对比，RAID 10 选择 2D+2D 的 RAID 方案，RAID 5 选择 3D+1P 的 RAID 方案，如图 2-30 和图 2-31 所示。

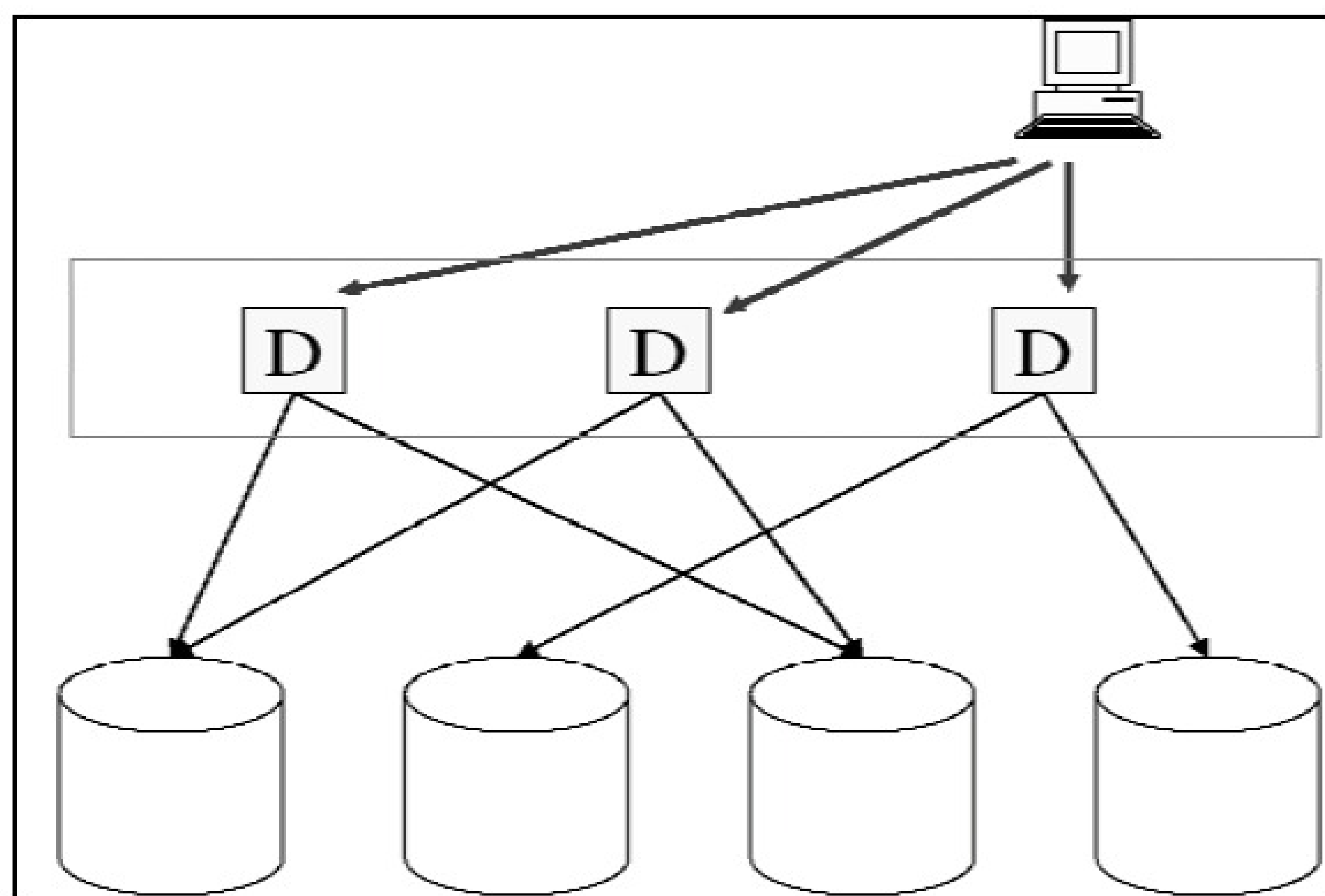


图 2-30　RAID 10(2D+2D)

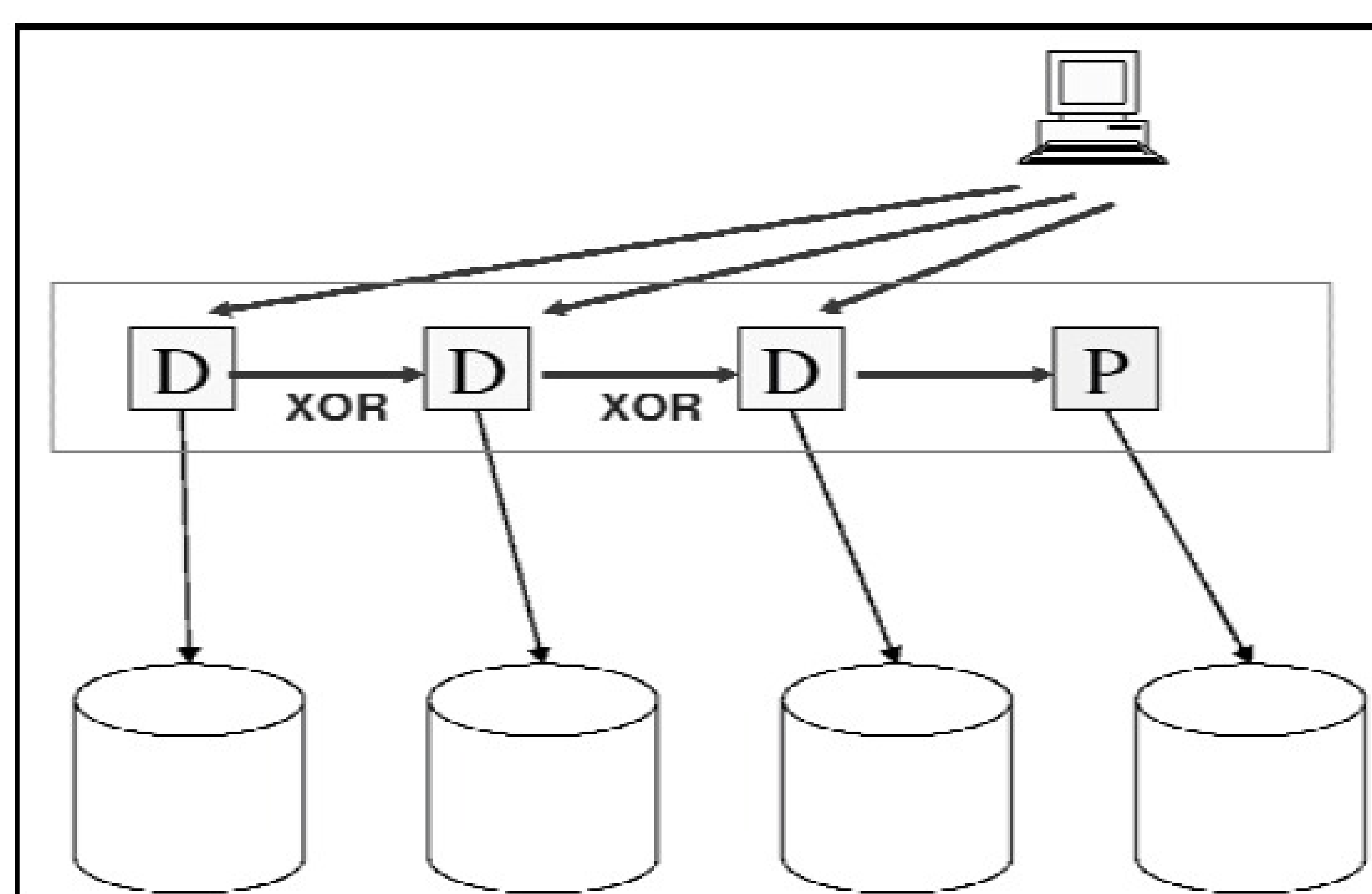


图 2-31　RAID 5(3D+1P)

1. 安全性方面的比较

其实在安全性方面，毋庸置疑，肯定是 RAID 10 的安全性高于 RAID 5。我们可以通过简单的分析来得出此结论。当有磁盘损坏时，对于 RAID 10，只有当这块磁盘对应的镜像盘也损坏，才会导致 RAID 失效。但是对于 RAID 5，剩下的 3 块盘中，任何一块盘出现故障，都将导致 RAID 失效。

在恢复的时候，RAID 10 恢复的速度也快于 RAID 5。

2. 空间利用率的比较

RAID 10 的利用率是 50%，RAID 5 的利用率是 75%。磁盘数量越多，RAID 5 的空间利用率越高。

3. 读写性能方面的比较

主要分析如下 3 个操作：读、连续写、离散写。

提示：

存储的 Cache 我们已经在 2.5.3 中讲述，因为这 3 个操作跟 Cache 有很大的关系。

1) 读操作方面的性能差异

RAID 10 可供读取有效数据的磁盘个数为 4，RAID 5 可供读取有效数据的磁盘个数也为 4(校验信息分布在所有的盘上)，所以两者在读方面的性能应该是基本一致的。

2) 连续写方面的性能差异

在连续写操作过程中，如果有写 Cache 存在，并且算法没有问题的话，RAID 5 比 RAID 10

甚至会更好一些，虽然也许并没有太大的差别(这里假定存储有一定大小，足够的写 Cache，而且计算校验的 CPU 不会出现瓶颈)。

因为这个时候的 RAID 校验是在 Cache 中完成的，如 4 块盘的 RAID 5，可以先在内存中计算好校验，然后同时写入 3 个数据+1 个校验。而 RAID 10 只能同时写入 2 个数据+2 个镜像。

如图 2-32 所示，4 块盘的 RAID 5 可以在同一时间写入 1、2、3 到 Cache，并且在 Cache 计算好校验之后，这里假定是 6，同时把这 4 个数据写到磁盘。而 4 块盘的 RAID 10 不管 Cache 是否存在，写的时候，都是同时写 2 个数据或 2 个镜像。

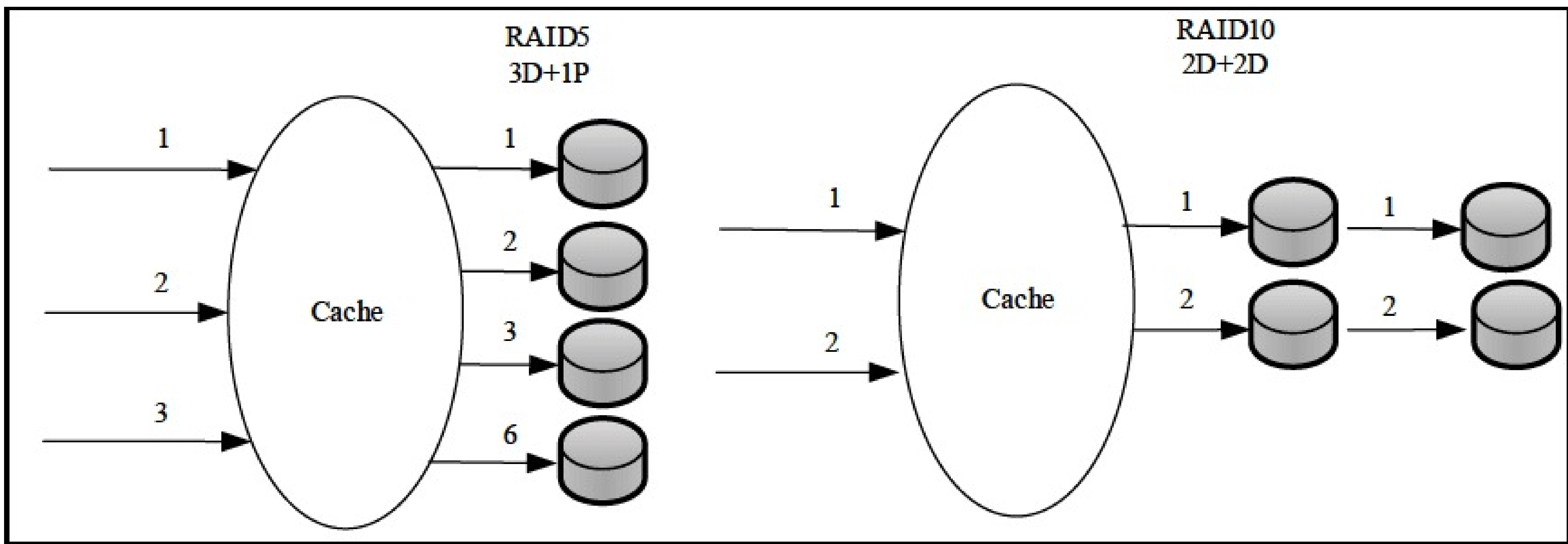


图 2-32　连续写

根据前面对缓存原理的介绍，写 Cache 是可以缓存写操作的，等到缓存写数据积累到一定时期再写到磁盘。但是，写到磁盘阵列的过程是迟早也要发生的，所以 RAID 5 与 RAID 10 在连续写的情况下，从缓存到磁盘的写操作速度会有较小的区别。不过，如果不是连续性的强连续写，只要不达到磁盘的写极限，差别并不是太大。

3) 离散写方面的性能差异

例如 DB2 数据库每次写一个数据块的数据，如 4KB 或 8KB，由于每次写入的量不是很大，而且写入的次数非常频繁，因此联机日志看起来会像是连续写。但是因为不保证能够填满 RAID 5 的一个条带，比如 32KB(保证每张盘都能写入)，所以很多时候更加偏向于离散写入(写入到已存在数据的条带中)。

离散写的时候，RAID 5 与 RAID 10 的工作方式有什么不同？查看图 2-33，假定要把数字 2 变成数字 4，那么对于 RAID 5，实际发生了 4 次 IO：先读出 2 与校验 6，可能发生读命中，然后在 Cache 中计算新的校验，写入新的数字 4 与新的校验 8。对于 RAID 10，同样的操作，最终 RAID 10 只需要两个 IO，而 RAID 5 需要 4 个 IO。

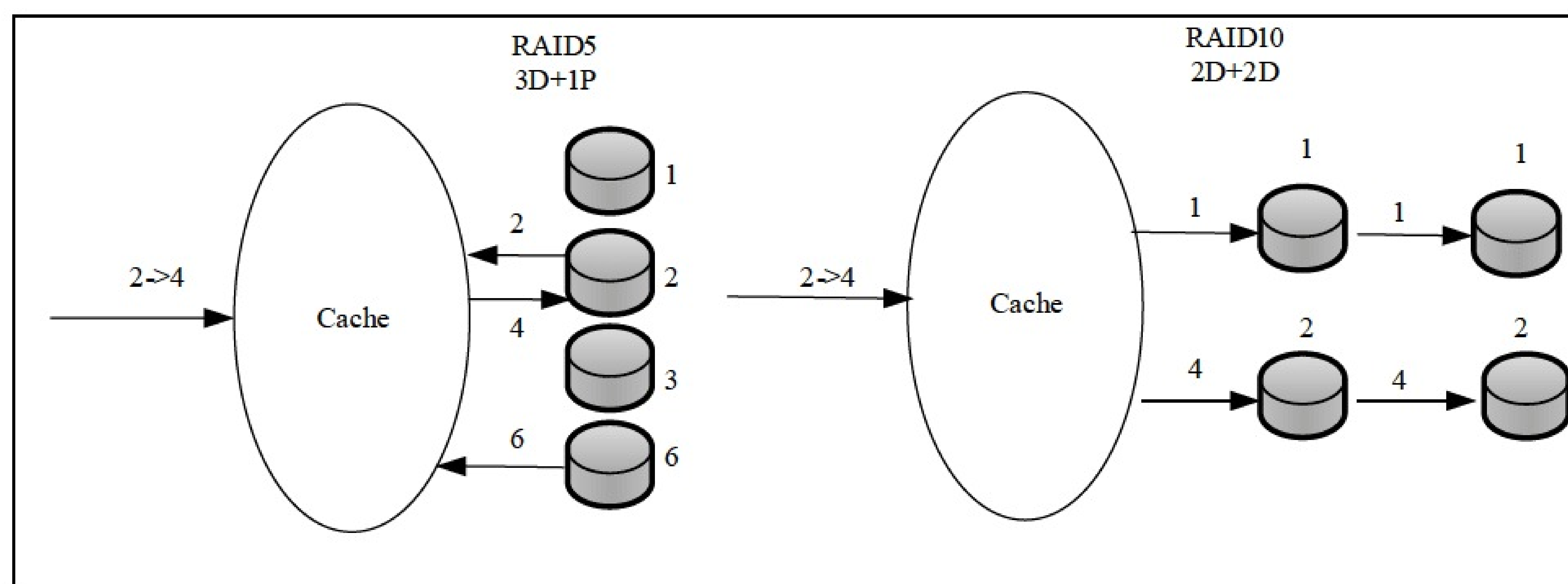


图 2-33　离散写

这里我忽略了 RAID 5 在那两个读操作的时候，可能会发生读命中操作的情况。也就是说，如果需要读取的数据已经在 Cache 中，可能是不需要 4 个 IO 的。这也证明了 Cache 对 RAID 5 的重要性，不仅仅是计算校验需要，而且对性能的提升尤为重要。

当然，并不是说 Cache 对 RAID 10 就不重要了，因为写缓冲、读命中等，都是提高速度的关键所在，只不过 RAID 10 对 Cache 的依赖性没有 RAID 5 那么明显而已。

4) 磁盘的 IOPS 对比

对空间利用率要求较高，而对安全性要求不是特别高的大文件存储系统，采用 RAID 5 比较好。相反，安全性要求很高，不计成本，小数据量频繁写入的系统采用 RAID 10 比较好。

根据我的经验与分析：小 I/O 的数据库类型操作，如 ERP 等应用，建议采用 RAID 10；而大型文件存储、数据仓库，如医疗 PACS 系统、视频编辑系统，从空间利用的角度，建议采用 RAID 5。

在一个实际的案例中，一个恢复压力很大的 standby(这里主要是写，而且是小 IO 的写)，采用了 RAID 5 的方案，发现性能很差，通过分析，每个磁盘的 IOPS 在高峰时期，快达到 200 了，导致响应速度非常慢。后来改造成 RAID 10，就避免了这个性能问题，每个磁盘的 IOPS 降到了 100 左右。所以，了解 RAID 5 和 RAID 10 原理，对于我们根据应用系统的特点来做存储 I/O 设计，从而保证性能非常重要。

2.8　良好存储规划的目标

良好存储规划的目标是从业务服务需求出发，尤其重点关注事务响应时间和事务处理的持续并发能力，存储系统资源规划以部件响应能力为基础，降低存储系统热点、容灾、

利用率及维护对服务能力的影响。

1) 降低主机磁盘响应时间

重点分析规划影响主机磁盘响应时间的关键因素，包括数据的 IO 特征、数据保护方式、部件利用率、数据处理并发性。

2) 降低资源热点对性能的影响

重点分析规划存储连接端口、缓存、磁盘、磁盘控制卡以及交换机端口，分析排查 IO 等待队列、Pending 数据、命中率等数据特点。

3) 降低容灾保护对性能的影响

重点分析规划容灾的高负载的数据卷、容灾数据卷的数据量及写数据的均衡能力，以及容灾数据处理的链路响应等。

4) 合理存储资源利用率定位

重点分析规划存储连接端口、缓存、磁盘、磁盘控制卡、容灾控制卡以及交换机端口，分析部件利用率运行状态，部件利用率直接影响数据处理响应时间。

5) 降低部件故障对系统性能的影响

重点分析规划存储连接端口、缓存、磁盘、磁盘控制卡以及交换机端口等系统部署冗余性，分析排查系统冗余部件的耦合性。

2.9　良好存储规划的设计原则

在进行存储 I/O 设计时，我们可以参考以下设计原则：

1) 确保您的数据均匀地分布在所有的物理磁盘中。如果您的数据仅位于几个盘中，那么使用多个逻辑单元号(LUN)或物理磁盘又有什么实际意义呢？

2) 如果您使用了 SAN 或其他类型的存储阵列，那么您应该尝试在创建阵列时使它们具有相同的大小和类型。您还应该在创建它们时，为每个阵列使用 LUN，然后将所有的逻辑卷分散到卷组中的所有物理卷上。

3) 您还应该确保镜像位于不同的磁盘和适配器。

4) 数据库索引表空间容器和数据表空间容器也应该位于不同的物理磁盘。

5) 对于表空间容器和数据库日志，应该存放到不同的 RAID GROUP 上。

6) 物理设备方面。使用高速适配器连接磁盘驱动器，具有大的 Cache，这一点是非常重要的，但是您必须确保总线本身不会成为瓶颈。要防止这种情况发生，确保将适配器分散到多个总线。同时，不要将过多的物理磁盘或 LUN 连接到任何一个适配器，因为这样

做也会对性能产生极大的影响。您配置的适配器越多越好，特别是在大量磁盘的利用率都很高的情况下。

7) 您还应该确保设备驱动程序尽可能支持多路径 I/O(MPIO)，MPIO 支持 I/O 子系统的负载平衡和流量控制。

8) 监控存储 Cache 的命中率，确保 Cache 的大小设置应该与业务特点(是 OLTP 还是 OLAP 业务类型)和数据页(data page)的大小设置合理。

9) 选择合适的 RAID 级别。根据自己应用系统的业务类型来选择合适的 RAID 级别。一般来说，有以下两种典型的业务类型：

- 70/30/50——70%读，30%写，Cache 命中率 50%，这是典型的高并发 OLTP 系统。
- 100/100——100%读，100%写，这是典型的 OLAP、DSS 系统。

对于小 I/O 的数据库类型操作，如 ERP、银行账务系统、移动计费系统等应用，建议采用 RAID 10；而大型文件存储、数据仓库，如报表系统、数据集市系统，从空间利用的角度，建议采用 RAID 5。

10) 在存储上选择合适的条带化大小(stripe size)，确保条带化大小要和数据库的 I/O 特点结合，因为对于 OLTP 应用，数据库 I/O 基本是随机小块读写；而对于 OLAP 应用，数据库 I/O 是连续大块读写。

11) 确保操作系统层面和存储层面的条带化大小设置一致，操作系统方面的条带化大小尽量大些。

2.10 存储相关性能调整案例

下面我们举一个实际的案例来说明在存储这个层面的优化对数据库性能提升的贡献。

客户应用系统环境：

- 存储：EMC 存储 VMAX
- 操作系统：IBM AIX
- 数据库：DB2 V9.7
- 多路径通道软件：POWERPATH

存储性能规划过程如图 2-34 所示。

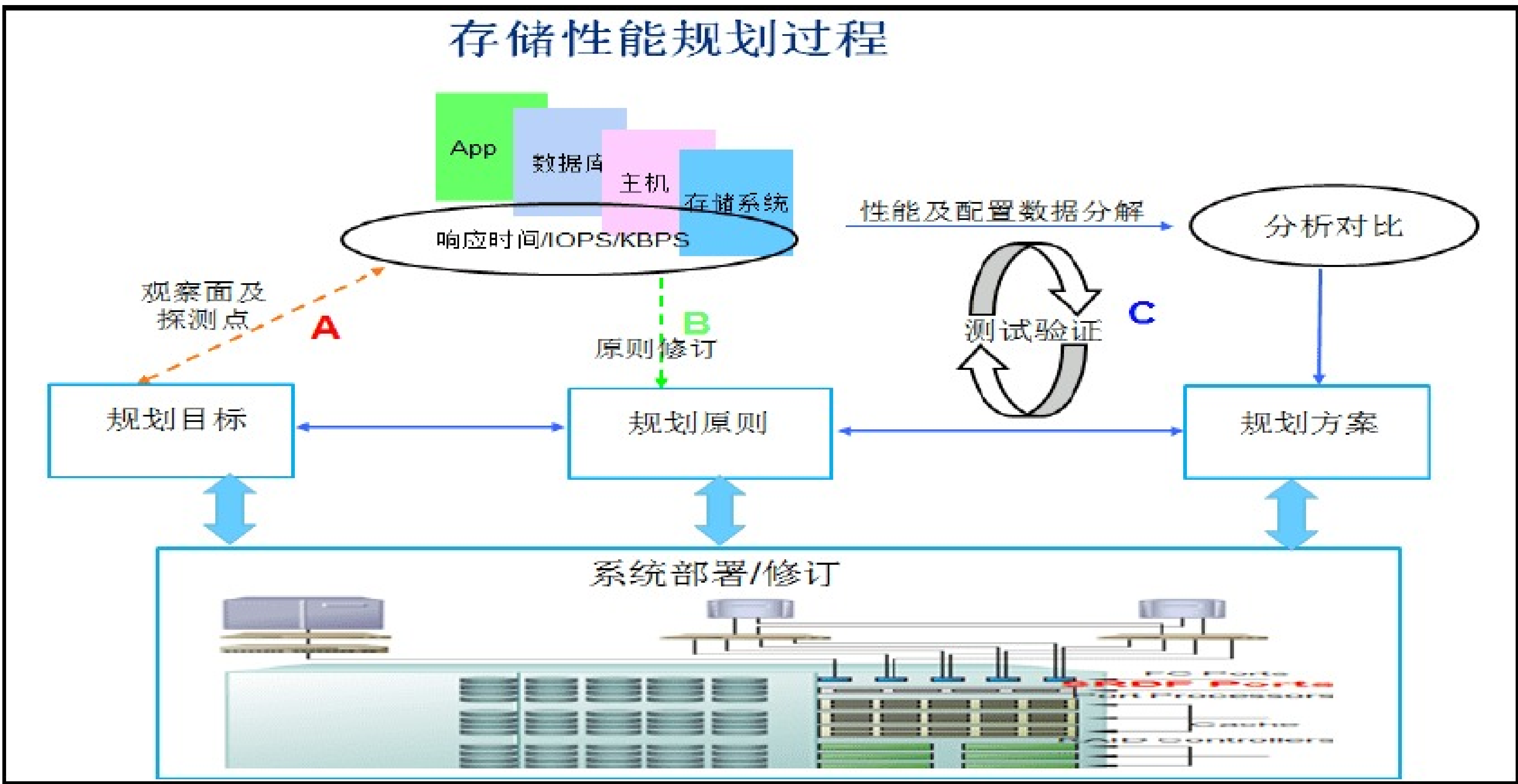


图 2-34　存储性能规划过程

B 系统经历多次压力测试，在存储调优之前，集中处理的大任务的执行时间有 85%左右在数据库中，性能的瓶颈出现在数据库层面。而且总的执行时间大幅度地超过了设计目标，某个大任务的执行时间达到了 8 小时。

经过多方深入分析，基本得出了结论：数据库本身的优化空间很小，主要的优化方向应集中在存储层次。

存储系统针对每次测试的特点，深入分析了存储系统存在问题，有针对性地对规划进行了修订，并对部署方案进行了调整，下面是比较重大调整的效果对比分析：

1) Host Stripe 调整效果分析

4 月 12 日，B 系统的日志空间修改为 Stripe 方式，Stripe Size 为 64KB，在未启用 SRDF 的情况下，主机使用 LOG 空间磁盘的写响应时间由 15ms 降低为 0.7ms，如图 2-35 所示。

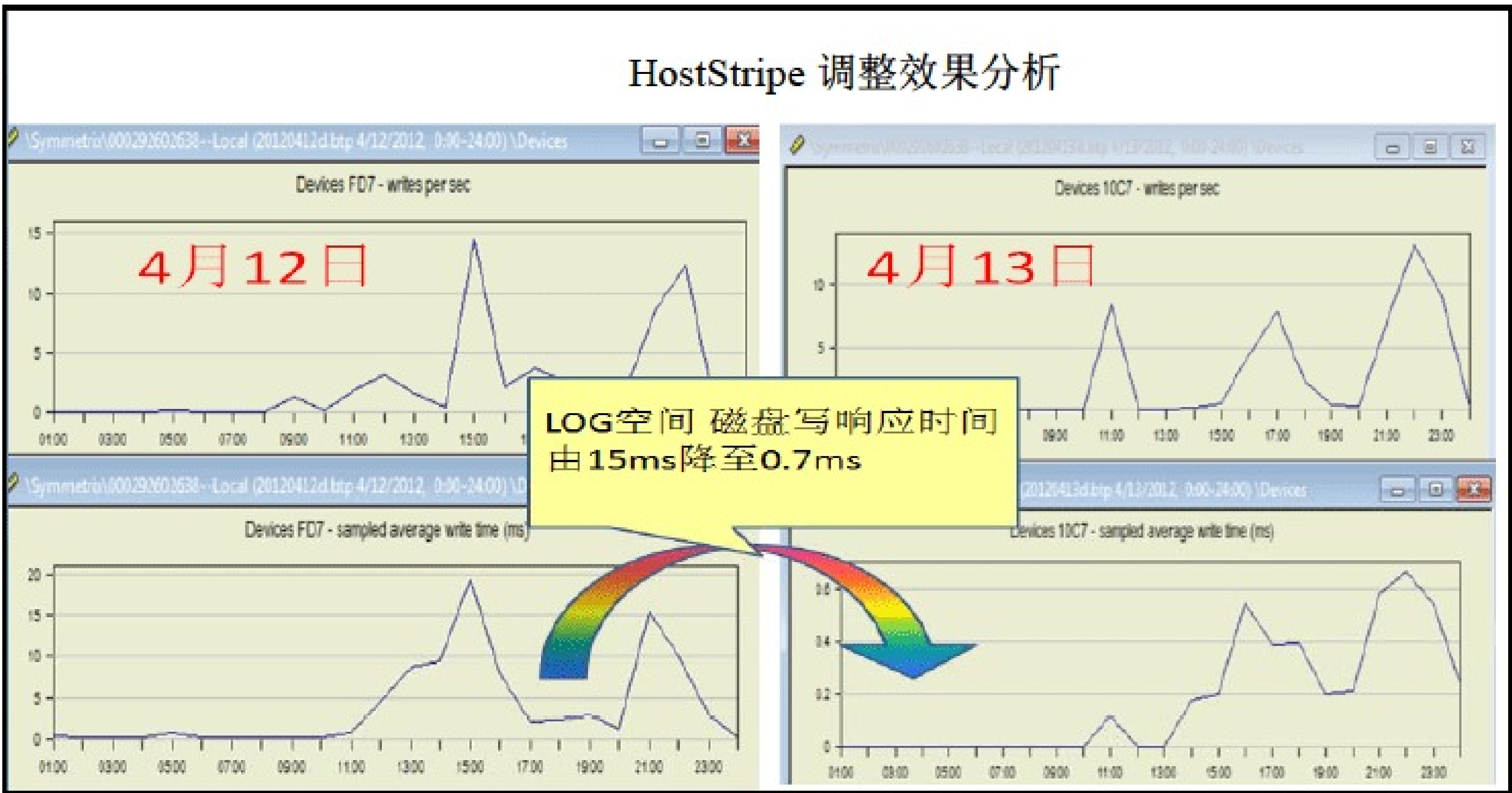


图 2-35　HostStripe 调整效果分析

2) FA 端口扩充效果分析

FA 端口经过多次扩充，4 月 25 日，B 系统的主机系统与存储系统使用 FA 关系为 1:1。在未启用 SRDF 的情况下，主机使用数据空间磁盘的写响应时间由 11ms 降低为 4.5ms，如图 2-36 所示。

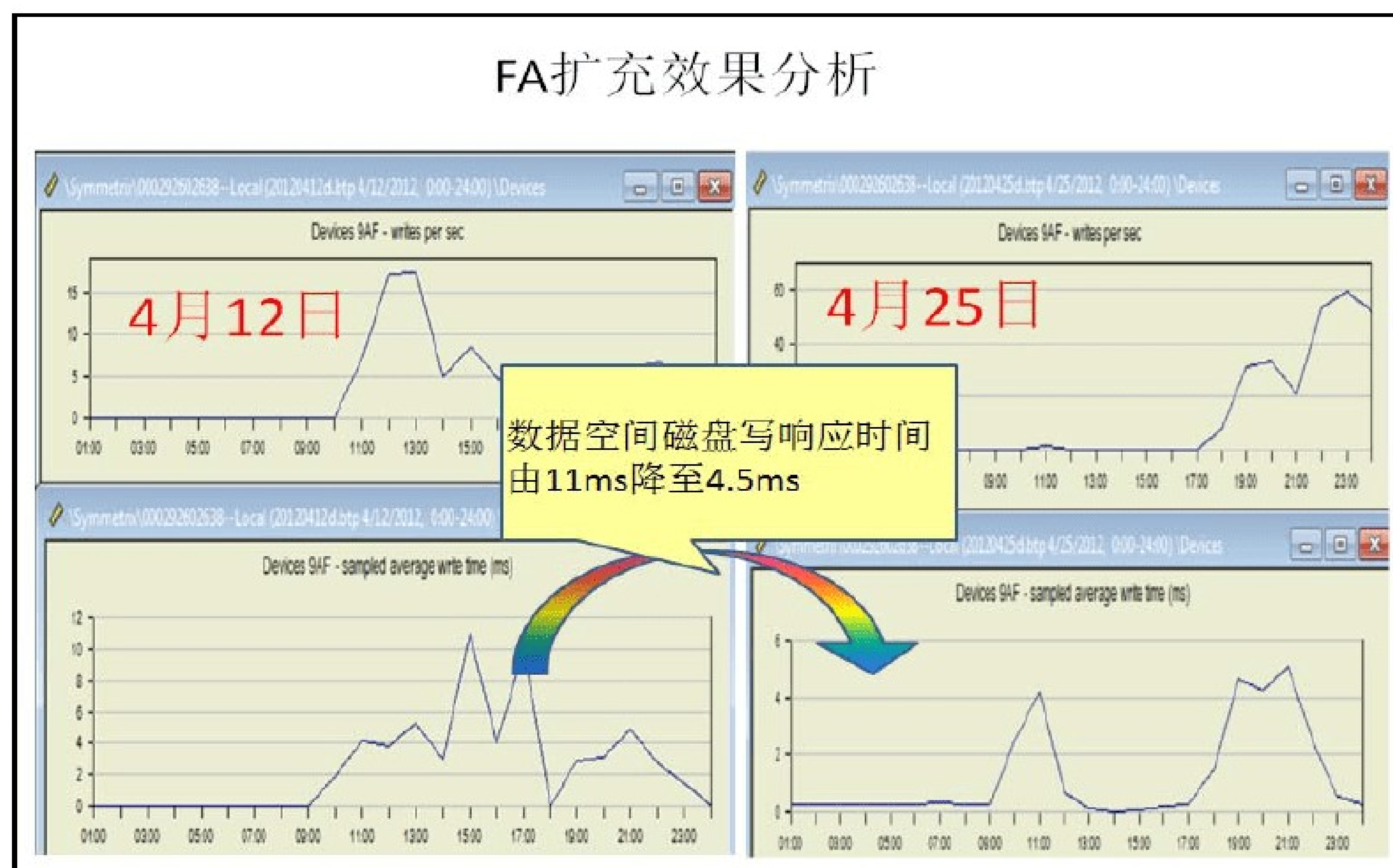


图 2-36　FA 扩充效果分析

3) 容灾端口扩充效果分析

RF 端口经过扩充，5 月 15 日，B 系统的主机系统与存储系统使用的 RF 端口数量由 4RF 扩充到 8RF。在未使用 Extanded License 和未加 BufferCredit 的情况下，主机使用数据空间磁盘的写响应时间由 69ms 降低为 45ms，如图 2-37 所示。

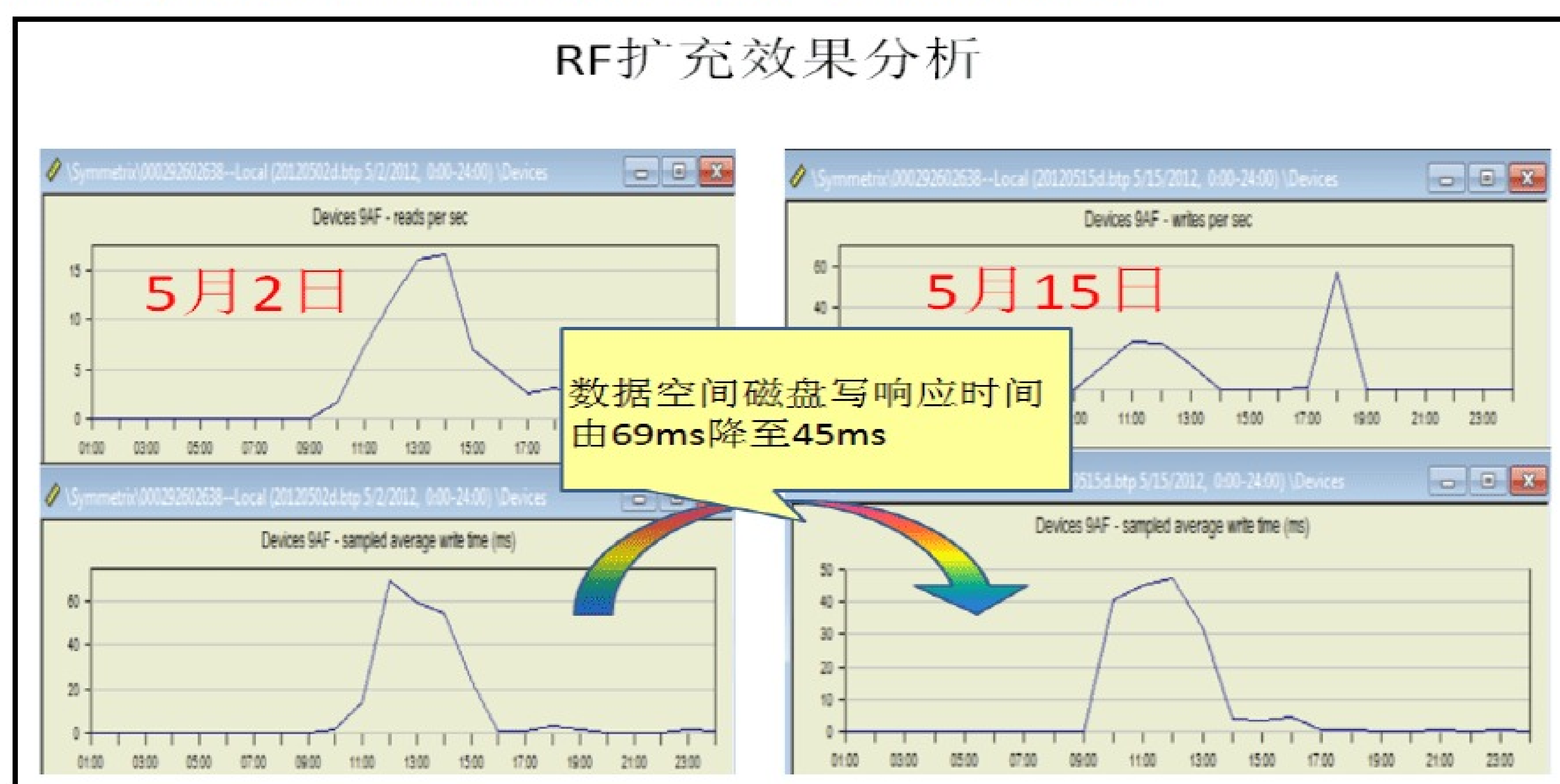


图 2-37　RF 扩充效果分析

4) 存储交换机增加 BufferCredit 及数据卷 Stripe 调整效果分析

5 月 31 日，存储交换机使用 Extanded License 和增加 BufferCredit=200，同时在 B 系统主机的 Data&Index 采用 HostStripe 为 64KB 的情况下，主机使用数据空间磁盘的写响应时间由 45ms 降低为 16ms，如图 2-38 所示。

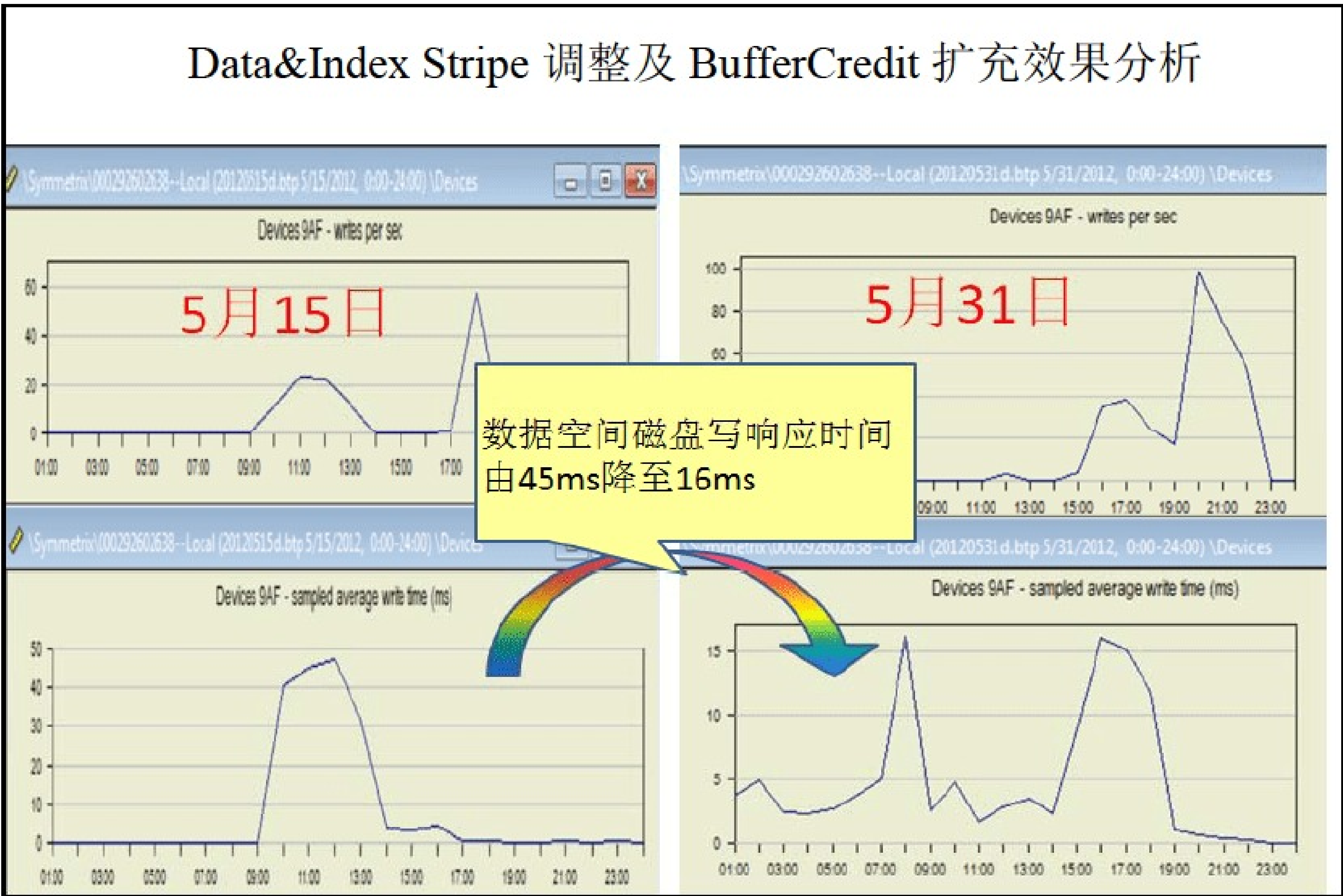


图 2-38　Data&Index Stripe 调整及 BufferCredit 扩充效果分析

小结：

从上述对比测试中我们可以看到，以上调整对性能的提升有极大帮助。随着存储层次的 I/O 时间大幅度缩小，业务的执行时间也有着非常大的变化，如批处理时间由原来的 8 小时缩短至不到 3 小时，这对性能的指标来说有着质的飞跃。数据库部分占用的处理时间比例也大幅度降低，数据库不再成为业务性能的瓶颈。

2.11　存储 I/O 性能调整总结

数据库中数据的访问离不开 I/O 行为，所以 I/O 的性能对数据库性能有着致命的影响，因此作为 I/O 行为的具体执行者——存储系统的性能就显得至关重要。而存储系统的特点又决定了事先的规划远远重要于事后的调优。

存储是一门学问，在本章中我们只是简单地讲解了存储的一些概念，以及存储 I/O 设计中的通用性能原则。为什么要讲解呢？这是因为在数据库的性能调优中，这一部分是必

须考虑的，而且往往会由于前期的存储 I/O 设计不合理而导致后期严重的性能问题。关于这一部分更深的知识，建议读者参考存储方面的书籍。

2.12 本章小结

性能调优是个系统工程，目前的业务系统都是一整套的复杂系统，从上至下包括以下几个层次：应用程序、中间件应用服务器、数据库、主机系统(操作系统)、光纤交换机和 SAN 存储网络。在系统发生性能问题时，性能问题的定位和调优很复杂。对于数据库的性能调优来讲，DBA 掌握操作系统的性能调优和存储 I/O 设计方面的调优知识，是非常有必要的。当然，操作系统和存储的 I/O 设计本章只是涉及了很小的一部分，更深入的知识还是需要读者参考相关的书籍。

第 3 章

DB2 性能监控

在《循序渐进 DB2(第 3 版)》一书中，我们给大家讲解了 DB2 数据库的主要监控工具，关于这些工具的原理机制读者可以参考该书。本章我们主要侧重于如何应用这些监控工具来定位常见的性能问题。

本章主要讲解如下内容：

- 快照监视器案例
- 事件监视器案例
- 利用表函数监控
- 性能管理视图监控案例
- db2pd
- db2mtrk

3.1 快照监视器案例

3.1.1 监控动态 SQL 语句

例 3-1 中显示的脚本将发出"db2 get snapshot for all on **dbname>snap.out**"命令，该命令包括"db2 get snapshot for dynamic SQL on **dbname** "命令的所有输出。如果您发现不能捕获很多的 SQL 语句，那么可以增加监控的时间。一条语句的输出的"Dynamic SQL Snapshot

Result" 部分看上去如例 3-1 所示。

提示：
为了保证监控尽可能多的 SQL 语句，建议增加 DBM 配置参数 mon_heap_sz。

例 3-1 动态 SQL 快照。

```
     Dynamic SQL Snapshot Result
 Database name                        = SAMPLE
 Database path                        = /db2/NODE0000/SQL00003/
 Number of executions      = 30
 Number of compilations               = 1  ----编译次数，是个累加值
 Worst preparation time (ms)          = 1624  -- SQL 语句编译的最长时间
 Best preparation time (ms)           = 1624
 Internal rows deleted                = 0
 Internal rows inserted               = 0
 Rows read                            = 1230
 Internal rows updated                = 0
 Rows written                         = 0
 Statement sorts                      = 0
Total execution time (sec.ms)         = 0.934186---SQL 语句的总执行时间，可以根
                                                   据这个时间排序找出执行时间
 Total user cpu time (sec.ms)         = 0.000000  --比较长的 SQL 语句
 Total system cpu time (sec.ms)       = 0.000000
 Statement text                       = select * from sales--------SQL 语句文本
   ..............................
```

在输出中可以搜索一些很有用的字符串。

"Number of executions" 可以帮助您找到应该调优的那些重要语句。它对于帮助计算语句的平均执行时间也很有用。

对于执行时间很长的语句，单独执行一次或许对系统要求不多，但是累加起来的结果就会大大降低性能。应尽量理解应用程序如何使用该 SQL，或许只需要对应用程序逻辑稍微重新设计一下就可以提高性能。

看看是否可以使用参数标记(parameter marker)，以便为语句创建包，这一点也很管用。参数标记可用作动态准备的语句(生成访问计划时)中的占位符。在执行的时候，就可以将值提供给这些参数标记，从而使语句得以运行。

有时候，使用 "grep" (UNIX)和 "findstr" (Windows)对快照输出执行初步的搜索非常方

便。如果发现了什么可疑的内容，就可以通过打开快照输出并找到问题所在，以便作进一步调查。

例如，为了识别是否存在死锁：

在 UNIX/Linux 下：

```
grep -n "Deadlocks detected" snap.out | grep -v "= 0" | more
```

在 Windows 下：

```
findstr /C:"Deadlocks detected" snap.out | findstr /V /C:"= 0"
```

再如，要搜索执行的最频繁的语句：

在 UNIX/Linux 下：

```
grep -n"Number of executions" snap.out | grep -v "= 0" | sort -k 6rn | more
```

在 Windows 下：

```
findstr /C: "Number of executions"  snap.out | findstr /V /C:"= 0"
```

注意：

在 Windows 下执行的时候，注意中英文转换，例如“Number of executions”在中文环境中应该用“执行次数”代替。这里主要讲思路。

"Rows read" 可帮助识别读取行数最多的动态 SQL 语句。如果读取的行数很多，通常意味着要进行表扫描。如果这个值很高，也可能表明要进行索引扫描，扫描时选择性很小，或者没有选择性，这跟表扫描一样糟糕。

您可以使用 Explain 工具来查看是否真的如此。如果有表扫描发生，那么可以对表执行一次 RUNSTATS，或者将 SQL 语句提供给 DB2 Design Advisor(db2advis)，以便令其推荐一个更好的索引，以此来进行弥补。如果是选择性很差的索引扫描，或许需要一个更好的索引。可以试试 Design Advisor。

```
grep -n"Rows read" snap.out | grep -v "= 0" | sort -k 5rn|more     ---UNIX/Linux
findstr /C:" Rows read" snap.out | findstr /V /C:"= 0"       ----Windows
```

"Total execution time"是将语句每次执行时间加起来得到的总执行时间。我们可以很方便地将这个数字除以执行的次数来得到平均执行时间。如果发现语句的平均执行时间很长，那么可能是因为表扫描和/或出现锁等待(lock-wait)。索引扫描和页面获取导致的大量 I/O 活动也是原因之一。通过使用索引，通常可以避免表扫描和锁等待。锁会在提交的时

候释放，因此如果提交得更频繁一些，或许可以弥补锁等待的问题。

在 UNIX/Linux 下：

```
grep -n "Total execution time" snap.out | grep -v "= 0.0"|sort -k 6rn| more
```

在 Windows 下：

```
findstr /C:"Total execution time" snap.out | findstr /V /C:"= 0.0" |sort /R
```

"Statement text"显示语句文本。如果注意到了重复的语句，这些语句除了 WHERE 子句中谓词的值有所不同以外，其他地方都是一致的，那么就可以使用参数标记，从而避免每次执行时重新编译语句。这样可以使用相同的包，从而帮助避免重复的语句准备，而这种准备的消耗是比较大的。还可以将语句文本输入到 Design Advisor (db2advis)中，以便生成最优的索引。

在 UNIX/Linux 下：

```
grep -E " Statement text"  snap.out | more
```

在 Windows 下：

```
findstr /C: "Statement text"  snap.out
```

3.1.2 监控临时表空间使用

在实际的 OLAP 生产环境中，我们看到数据库的系统临时表空间使用非常频繁，而且此时数据库性能不佳，那么我们如何查看各个应用对 DB2 临时表空间的使用情况呢？我们都知道应用程序在 DB2 上运行的时候，会产生临时表。这些操作包括：sort、group、build index 和 reorg 等。如果想要监视各个应用程序对临时表空间的使用情况，可以使用下面的步骤：

(1) 打开 monitor switches 中的 table 监视器：

```
db2 update monitor switches using table on
```

(2) 对数据库抓取 table 快照：

```
db2 get snapshot for tables on <db_name>
```

(3) 在快照的输出中定位到系统临时表的信息：

```
Table Name = TEMP (00013,00002)
```

这样的信息表示这个表是系统临时表，括号中的第一个数值是临时表空间 ID，第二个

数值是 DB2 分配给这个临时表的临时 ID。

(4) 在系统临时表中定位对应的 agent ID：

```
Table Schema = <984><BR12ADM > ------使用临时表空间的 agent ID
Table Name = TEMP (00013,00002)
Table Type = Temporary
Data Object Pages = 104
Rows Read = 420
Rows Written = 827
Overflows = 0
Page Reorgs = 0
```

其中，Table Schema 后面的第 1 个数字就是这个临时表归属的 agent ID.

(5) 通过这个 agent ID，我们就可以使用应用程序快照或 db2pd 等工具找到对应的应用：

```
db2 get snapshot for application agentid 984
..................略....................................
分块游标                                    = NO
动态 SQL 语句文本:
select job,sex,workdept,count(*) from employee
group by cube(job,sex,workdept)
```

通过这种方式，我们可以定位哪条 SQL 语句使用临时表空间最多，然后尝试对这条 SQL 语句进行调整。具体调整的方法可以参见“第 9 章：SQL 语句调优”。

在实际的环境中，往往通过手动抓取很难抓取到具体的语句，我们可以尝试着在监控系统中调用一些我们预先写好的脚本去抓取，这样在遇到问题进行查看时就很方便了。

如下是抓取临时表空间使用的范例脚本，请大家参考：

```
sh gettmpsql.sh dbname

功能：找到数据库中占用临时表空间最大的 10 条 SQL 语句。


#/bin/ksh!
#
db=$1
#
db2 connect to $db
#
db2 get snapshot for tables on $db > $db.snap
#
cat $db.snap |grep -i -p "Table Name         = TEMP"> 1.db2
```

```
    cat 1.db2 |grep -i "Data Object Pages"|sort -k 5rn|awk '{print $5}'|head
-10 > tab.list
    #
    today='date +%Y%m%d%H%M'
    echo "Begin checking" > $db $today.log
    #
    cat tab.list|while read line
       do
       agent='cat 1.db2 |grep -p $line |grep -i "Table Schema"|awk '{print
$4}'|sed  's/<//g'|sed  's/>/ /g'|awk '{print $1}''
       echo
"*********************************************************************
***********************" >>$db $today.log
       cat 1.db2 |grep -p $agent >> $db $today.log
       echo "^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^" >>$db $today.log
       echo "Agent $agent INFO:" >>$db $today.log
       db2pd -d $db -apinfo $agent |grep -i -p "Statement :" >>$db $today.log
       echo
       "*******************************************************************"
       >>$db $today.log
       done
    rm 1.db2 tab.list
    more $db_$today.log
```

输出信息类似如下所示：

```
    *********************************************************************
     Table Schema        = <42021><UPERTAY >
     Table Name          = TEMP (00004,00004)
     Table Type          = Temporary
     Data Object Pages    = 6580
     Rows Read           = 0
     Rows Written         = 13159
     Overflows           = 0
     Page Reorgs          = 0

     Table Schema        = <42021><UPERTAY >
     Table Name          = TEMP (00004,00005)
     Table Type          = Temporary
     Data Object Pages    = 6587
     Rows Read           = 5015
     Rows Written         = 13173
     Overflows           = 0
     Page Reorgs          = 0
```

```
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    Agent 42021 INFO:
      List of active statements :
        UOW-ID :          1
        Activity ID :     1
        Package Schema :  NULLID
        Package Name :    SYSSH200
        Package Version :
        Section Number :  1
        SQL Type :        Dynamic
        Isolation :       CS
        Statement Type :  DMUL, Select (blockable)
        Statement :       select
        a.ZHDM,a.KHDM,b.BRLHZ,b.BYLHZ,b.BJLHZ,b.BNLHZ from
upeotay.SL_YW_ZTXX_DS a inner join (select * from upeotay.SL_SJ_PTCPLHZ_
BBDS TEMP where SJRQ=:L0 ) b on b.ZTDM=a.ZHDM where a.SL CPDM=:L1 and a.KHRQ<=:L2
    *****************************************************************
```

3.2　事件监视器及监控案例

快照监视器监控的是数据库的实时数据，Event Monitor(事件监视器)记录某事件(event)或转变(transition)出现时某段时间内数据库的活动情况。事件监视器收集监视器数据，例如特定事件或事务发生。因此，事件监视器提供了当事件或活动发生的时候，不能使用快照监视器监视的收集数据库系统监视器数据的方法。

例如，假设您想要捕获每当死锁周期发生时的监视器数据。如果对死锁的概念比较熟悉的话，您就应该知道数据库有被称为死锁检测器的特殊线程(db2dlock)，该线程在后台安静地运行，并且在预定的间隔时间(dlchktime)内会“苏醒”，扫描当前正在锁定的系统。如果在死锁周期内发现死锁，死锁检测器将会随机选择、回滚并且终止涉及在此次周期中的任意事务。被选择出来的那个事务将会接收到 SQL 错误代码(-911)，并且所有实际上已经获得的锁被释放，以便其他的事务能够继续执行。

像这样一系列的事件信息不能被快照监视器捕获，当死锁出现时，DB2 将通过对多个事务中的某个事务发出 ROLLBACK(回退)操作去解决死锁问题。有关死锁操作的信息不容易用 Snapshot Monitor 捕获到，因为在能够拍得快照之前，死锁可能已经被解决了。然

而，事件监视器却可以捕获该事件的重要信息，因为它可以在死锁周期被检测到的瞬间被激活。

这两种监视器的另外一个显著的不同是：快照监视器以后台进程的方式驻留，从数据库连接建立就开始捕获监视器数据；相反，事件监视器必须在它们使用之前专门去建立和激活。几个不同的事件监视器可以共存，并且每个事件监视器只有在特定类型的事件或事务发生的时候才会被激活。

Event Monitor 类似其他的数据库对象，因为它们是使用 SQL DDL(数据定义语言)创建的，主要的差别是 Event Monitor 可以像 Snapshot Monitor 的开关那样被打开或关闭。

当创建 Event Monitor 时，必须声明要被监视事件的类型。当以下事件发生时，相应事件记录便被记录下来：

DATABASE——当最后一个应用程序与数据库断开时，记录事件记录。

TABLES——当最后一个应用程序与数据库断开时，记录每个活动表的事件记录。

DEADLOCKS——对于每个死锁，记录事件记录。

TABLESPACES——当最后一个应用程序与数据库断开时，对每个活动表空间记录事件。

CONNECTIONS——当应用程序与数据库断开时，对每个数据库连接事件记录事件。

STATEMENTS——对于由应用程序发出的每条 SQL 语句(动态或静态)，记录事件。

TRANSACTIONS——当事务(COMMIT 或 ROLLBACK)完成时，为该事务记录事件。

Event Monitor 的输出存放在目录或命名管道中(也可以存放在表中，这些事件监控的表可以自动生成)。当 Event Monitor 被激活时，将生成管道和文件。如果 Event Monitor 的目标位置是命名管道，那么可以通过应用程序从管道读出数据。如果 Event Monitor 的目标位置是目录，数据流将被写入一系列文件中。这些文件顺序编号并且都有文件附加名 evt(例如 00000000.evt、00000001.evt 等)。当定义事件监视器时，规定 Event Monitor 事件文件的大小与数目。

事件监视器的创建方法和步骤如下：

(1) 创建 SQL Event Monitor，写入文件：

```
db2 "create event monitor <evmname> for <evntype> write to file 'directory'"
```

例如：

```
db2 "create event monitor SQLCOST for deadlocks write to file '/tmp/event'"
```

(2) 激活事件监视器(确保有充足的可用磁盘空间)：

```
db2 "set event monitor SQLCOST state = 1"
```

(3) 让应用程序运行。

(4) 取消激活事件监视器：

```
db2 "set event monitor SQLCOST state = 0"
```

(5) 使用 DB2 提供的 db2evmon 工具来格式化 SQL Event Monitor 原始数据(根据 SQL 吞吐率可能需要数百兆字节的可用磁盘空间)：

```
db2evmon -db DBNAME -evm SQLCOST > sqltrace.txt
```

(6) 浏览整个已格式化的文件，寻找显著大的成本数(是个耗时的过程)：

```
more sqltrace.txt
```

其实在实际的性能调优中，事件监视器并没有被广泛运用，因为 DB2 的事件监视器在监控期间会产生非常大的数据。一般在频繁发生相关问题的时候才打开抓取一段时间内的事件。

事件监控器案例

下面我们举一个事件监视器存放在表中的例子：对于数据库管理员，调优数据库常常是一项挑战。调优应用程序是一种方法，但在大多数生产系统中 DBA 很少甚至不能更改源代码，因此限制了他们调优应用程序的能力。这在 DBA 使用第三方工具时尤为如此。所以，通常最有效的调优方法是解决问题的根源，即从 SQL 语句本身入手。通常通过查找哪些 SQL 语句消耗的资源最多来获得最佳性能，然后决定采取一定的措施来减少资源消耗。

通常，在第一次安装数据库时，会将其性能调至最优，但随着时间的流逝，一些常用的东西开始变得越来越慢。这在拥有大量数据的系统(例如决策支持系统)中尤为明显。用户开始看到如锁升级、全表扫描以及排序这样的因素造成性能下降，这些操作往往会迫使系统访问磁盘而不是访问内存。当出现这种情况时，最好检查一下 SQL，看看可以做哪些改进。

我们举这个事件监控器案例，旨在解决的问题是：针对通常的活动，调优应该用于最频繁访问数据库的 SQL 以及最消耗资源的 SQL。为了简化如何监控应用程序中 SQL 语句的问题，我们通过 DB2 的事件监视器，确定哪些 SQL 语句消耗的资源最多。

1) 利用表的方式

首先，必须创建事件监视器，运行监视器来收集将要分析的数据。

(1) 创建事件监视器。打开新的 DB2 命令行处理器会话，然后执行以下 DB2 UDB 命令：

```
db2 update monitor switches using statement on
db2 "create event monitor sql_trace for statements write to table"
```

创建完之后，我们可以看到，DB2 自动生成了下面这张表：

```
db2 list tables for all|grep -i stmt
STMT_SQL_TRACE                DB2INST1       T 2012-11-09-11.18.44.630980
db2 set event monitor sql_trace state=1
```

(2) 使该会话一直处于打开状态，直到这些数据库活动完成。确保 STMT_SQL_TRACE 所在的表空间有足够的存储空间，在此时间有可能会产生非常大量的数据。表空间的大小取决于用户想要捕获的 SQL 语句的数目和事务量。

(3) 执行正常的数据库活动，直到您想要监控的时段结束。这一监控阶段可以是问题产生时期，也可以是通常的数据库高峰期间。

(4) 回到在步骤(1)中打开的会话，然后发出以下命令：

```
db2 set event monitor sql_trace state=0
db2 terminate
```

2) 分析输出

由于已经在数据库表 STMT_SQL_TRACE 中存储了所需要的信息，因此可以查询该表以确定“讨厌”且耗时的 SQL 语句。

需要确定 4 类 SQL 语句。在执行以下查询以确定这些语句之前，在数据库配置参数中，至少要为应用程序堆大小(applheapsz)分配 256 个页面。

- 按照执行时间降序排列执行耗时最长的 SQL 语句。为了确定这些语句，使用下面的 SQL SELECT 语句：

```
    select stmt_text, (stop_time-start_time) as ExecutionTime  from
stmt_sql_trace  where stmt_operation not in (7,8,9,19)  order by decimal
(ExecutionTime) desc fetch first 10 rows only
```

- 按照频率降序排列执行次数最多的 SQL 语句。可以用下面这条查询来确定这些语句：

```
    select distinct varchar(stmt_text,2000) as stmt_text,count(*) Count from
stmt_sql_trace where stmt_operation not in (7,8,9,19) group by
varchar(stmt_text,2000) order by Count desc  fetch first 10 rows only
```

- 按照 CPU 时间降序排列最耗 CPU 时间的 SQL 语句。可以用下面这条查询来确定这些语句：

```
    select stmt_text ,user_cpu_time as UserCPU from stmt_sql_trace where
stmt_operation not in (7,8,9,19) order by usercpu desc fetch first 10 rows only
```

- 按照总排序时间降序排列排序时间最长的 SQL 语句。可以用下面这条查询找到这些语句：

```
    select stmt text ,total sort time as TotalSortTime from stmt sql trace
where stmt operation not in (7,8,9,19 ) order by decimal(total sort time) desc
fetch first 10 rows only
```

捕获每一类中的 SQL 语句，并将它们放在 tune.sql 文件中。将下面这行插入到该文件中，这样可以更改工作负载中每条语句的执行频率：

```
--#SET FREQUENCY <x>
```

这里的<x>表示随后要执行 SQL 语句的次数。您的 tune.sql 文件类似于下面这样：

```
--#SET FREQUENCY 100
SELECT COUNT (*) FROM EMPLOYEE;
SELECT * FROM EMPLOYEE WHERE LASTNAME='HAAS';
--#SET FREQUENCY 1
SELECT AVG (BONUS), AVG (SALARY) FROM EMPLOYEE
GROUP BY WORKDEPT ORDER BY WORKDEPT;
```

将这些 SQL 语句复制到 tune.sql 之后，检查任何 SQL 语句的 WHERE 子句中是否具有参数标志符(?)。将参数标志符改为适当的数据类型值，以便在没有任何错误的情形下执行 SQL 语句。

为了确定哪些索引可能提高性能，按如下所示执行索引顾问程序：

```
db2advis -d sample -i tune.sql -t 0 -o tuneidx.sql
```

所有推荐的索引将放置在文件 tuneidx.sql 中。编辑该文件，在文件开始处添加一条连接语句：

```
connect to <sample> user <userid> using <password>;
```

在该文件末尾添加下面这行：

```
terminate;
```

现在可以运行该文件以创建推荐的索引：

```
db2 -tvf tuneidx.sql -z tuneidx.log
```

提示：

其中，tuneidx.log 捕获 tuneidx.sql 的所有输出。

另外，对于运行性能较差的 SQL 语句，我们还可以使用如下 shell 脚本来分析 DB2 中的执行时间和返回的条数，从这些信息使我们可以找到当前数据库中性能较差的语句：

```
监控某段时间内数据库执行语句的性能
使用： sh getlongsql.sh <dbname> <interval(S)>
#!/usr/bin/ksh
#shell script name is getlongsql.sh
if [ $# -eq 0 ] ; then
 echo "Usage:getlongsql.sh <database1> <interval>"
 exit
fi
DB=$1
INTERVAL=$2
FILENAME=sql.txt
connectdb()
{
db2 GET CONNECTION STATE|grep 'Connected'>/dev/null
res2=$?
if [[ $res2 -eq 0 ]]
then
  echo "The DB $DB has been connected!"
else
  db2 connect to $DB
  res=$?
  i=0
  while [ $i -lt 3 ]
  do
         i='expr $i + 1'
                if [[ $res -eq 0 ]]
                then
                  echo "The DB $DB has been connected!"
                  break
                else
                  sleep 3
                  db2 connect to $DB
```

```
                    res=$?
                  fi
     done
    fi
    }
    openEventMonitor()
    {
    db2 "drop table STMT_SQLMON";
    db2 "drop table CONTROL_SQLMON";
    db2 "drop table CONNHEADER_SQLMON";
    db2 "drop event monitor SQLMON";
    db2 "create event monitor SQLMON for statements write to table";
    db2 "set event monitor SQLMON state 1"
    sleep $INTERVAL
    db2 "set event monitor SQLMON state 0"
    db2 "select START TIME, STOP TIME, (STOP TIME-START TIME) as
dur,ROWS READ,FETCH COUNT,substr(stmt text,1,400) as stmt from STMT SQLMON
order by dur desc" > $FILENAME
    more $FILENAME
    }
    connectdb
    openEventMonitor
```

3.3　利用表函数监控

DB2 数据库提供了很多表函数，通过这些表函数可以获得很多与数据库性能有关的信息，所以也可以通过使用这些表函数代替 GET SNAPSHOT 命令来收集这些监控数据。

快照表函数能够捕获的许多监视器元素都受控于监视器开关。如果快照表函数中的某些函数描述中提到特定监视器开关，就表明受控于该开关。

我们可以通过引用 20 多个可用的快照监视器表函数中的一个来构造 SQL 查询以收集快照数据。表 3-1 列出了这些表函数并指明它们所能获取的具体快照信息。

表 3-1　部分表函数列表

快照表函数	返回的信息
MON_GET_INSTANCE	数据库管理器信息
MON_GET_AGENT	返回代理程序信息
MON_GET_CONTAINER	返回表空间容器信息
MON_GET_TABLESPACE	返回表空间的信息
MON_GET_DATABASE	数据库信息。只有当至少有一个应用程序连接至数据库时，才会返回信息
MON_GET_CONNECTION	连接至分区上数据库的应用程序上的有关锁等待的应用程序信息。这包括累积计数器、状态信息和最近执行的 SQL 语句(假定设置了语句监视器开关)
MON_GET_UNIT_OF_WORK	每个连接至分区上数据库的应用程序的常规应用程序标识信息
MON_GET_APPL_LOCKWAIT	有关锁等待连接至分区上数据库的应用程序的应用程序信息
MON_GET_ACTIVITY	有关连接至分区上数据库的应用程序的语句的应用程序信息，这包括最近执行的 SQL 语句(假定设置了语句监视器开关)
MON_GET_TABLE	连接至数据库的应用程序所访问的每个表的表活动信息。需要表监视器开关
MON_GET_LOCKS	数据库级别上的锁信息，以及每个连接至数据库的应用程序在应用程序级别上的锁信息。需要锁监视器开关
MON_GET_BUFFERPOOL	指定数据库的缓冲池活动计数器。需要缓冲池监视器开关
MON_GET_PKG_CACHE_STMT	来自用于数据库的 SQL 语句高速缓存的某个时间点语句信息

我们可以通过下面的 SQL 语句返回所有的表函数：

```
db2 "select distinct funcname from syscat.functions where funcname like
```

```
'MON GET%'"
    FUNCNAME
    -----------------------------------------------------------------------
    MON GET ACTIVITY
    MON_GET_ACTIVITY_DETAILS
    MON GET AGENT
    MON GET APPLICATION HANDLE
    MON GET APPLICATION ID
    MON GET APPL LOCKWAIT
    MON GET AUTO MAINT QUEUE
    MON GET AUTO RUNSTATS QUEUE
    MON GET BUFFERPOOL
    MON GET CF
    MON GET CF CMD
    MON GET CF WAIT TIME
    MON GET CONNECTION
    MON GET CONNECTION DETAILS
    MON GET CONTAINER
    MON GET DATABASE
    MON GET DATABASE DETAILS
    MON GET EXTENDED LATCH WAIT
    MON GET EXTENT MOVEMENT STATUS
    MON GET FCM
    MON GET FCM CONNECTION LIST
    MON GET GROUP BUFFERPOOL
    MON GET HADR
    MON GET INDEX
    MON GET INDEX USAGE LIST
    MON_GET_INSTANCE
    MON GET LATCH
    MON GET LOCKS
    MON GET MEMORY POOL
    MON GET MEMORY SET
    MON GET PAGE ACCESS INFO
    MON GET PKG CACHE STMT
    MON GET PKG CACHE STMT DETAILS
    MON GET QUEUE STATS
    MON GET REBALANCE STATUS
    MON GET ROUTINE
    MON GET ROUTINE DETAILS
    MON GET ROUTINE EXEC LIST
    MON_GET_RTS_RQST
```

```
MON GET SECTION
MON GET SECTION OBJECT
MON GET SECTION ROUTINE
MON GET SERVERLIST
MON GET SERVICE SUBCLASS
MON GET SERVICE SUBCLASS DETAILS
MON GET SERVICE SUBCLASS STATS
MON GET SERVICE SUPERCLASS STATS
MON GET TABLE
MON GET TABLESPACE
MON GET TABLESPACE QUIESCER
MON GET TABLESPACE RANGE
MON GET TABLE USAGE LIST
MON GET TRANSACTION LOG
MON GET UNIT OF WORK
MON GET UNIT OF WORK DETAILS
MON GET USAGE LIST STATUS
MON GET UTILITY
MON GET WORKLOAD
MON GET WORKLOAD DETAILS
MON GET WORKLOAD STATS
MON GET WORK ACTION SET STATS

  61 record(s) selected.
```

快照监视器数据组织

所有的快照表函数都返回一张监视器数据表，其中的每一行代表一个正被监控的数据库对象实例，而每一列代表一个监视器元素。监视器元素代表数据库系统状态的特定属性。

捕获监视器数据快照

要使用快照表函数捕获直接访问的快照，请完成以下步骤：

(1) 连接至数据库。可以是您需要监控的实例中的任何数据库。要能够使用快照表函数发出 SQL 查询，您必须连接至数据库。

例如：

```
db2 connect to sample
```

(2) 确定您需要捕获的快照类型，以及您需要监控的数据库和分区。除了收集这个信息之外，请打开任何可应用的监视器开关(通过检查表 3-1 中的快照表函数描述可以确定这一点)。

例如，如果您想要捕获表活动数据的快照(使用表函数 SNAPSHOT_TABLE)，那么将

需要激活 TABLE 监视器开关：

```
db2 update dbm cfg using DFT_MON_TABLE on
```

(3) 使用期望的快照表函数发出查询。

例如，以下查询捕获有关当前已连接分区的 SAMPLE 数据库的表活动信息的快照：

```
db2 "select * from TABLE(MON_GET_TABLE('','',-2)) as T"
```

表函数有三个输入参数：

- VARCHAR(128)，用于 SCHEMA 名称。如果输入 NULL，就返回所有 SCHEMA 对应的表信息。
- VARCHAR(128)，用于表名称。如果输入 NULL，就返回所有表信息。
- SMALLINT，用于分区号。对于分区号参数，输入整数(0 到 999 之间的值)以对应您需要监控的分区号。要捕获当前已连接分区的快照，请输入值 - 1 或 NULL。要捕获分区数据库所有分区快照，请输入值 - 2。

使用下面的语法将会创建引用非数据库管理器级表函数的查询：

```
SELECT * FROM TABLE(<FunctionName>(<DBName>,<PartitionNum>)) AS CorrelationName
```

如果想要通过使用快照监视器的表函数 MON_GET_LOCKS 来抓取当前连接的数据库的锁定数据的信息，可以执行下面的语句：

```
SELECT * FROM TABLE(MON_GET_APPL_LOCKWAIT(NULL, -2)) AS LOCK_INFO
```

如果我们使用 SAMPLE 数据库(先前的例子)作为当前被连接的数据库，那么执行 GET SNAPSHOT FOR LOCKS ON SAMPLE 命令返回的信息将会与先前表函数监控非常相似。

3.4 性能管理视图及案例

DB2 提供了很多性能管理视图(这些性能管理视图类似 Oracle 数据库中的以 v$开头的动态性能视图)，使用这些管理视图可以获得与表函数和快照类似的监控数据。表 3-2 列出了部分管理视图，需要注意的是这些视图的模式名都是 SYSIBMADM。我们可以使用 db2 list tables for schema sysibmadm 命令来获取所有的性能管理视图。

同样，在这些视图中能够获得哪些监控数据很多取决于监控开关的设置情况。

表 3-2　DB2 的部分管理视图

视　图　名	模　式　名	描　　述
APPLICATIONS	SYSIBMADM	数据库中运行的应用
APPL_PERFORMANCE	SYSIBMADM	每个应用中 rows selected 与 rows read 的比率
BP_HITRATIO	SYSIBMADM	缓冲池的命中率
BP_READ_IO	SYSIBMADM	缓冲池读的信息
BP_WRITE_IO	SYSIBMADM	缓冲池写的信息
CONTAINER_UTILIZATION	SYSIBMADM	表空间中容器的利用率信息
LOCKS_HELD	SYSIBMADM	当前获得的锁的信息
LOCKWAITS	SYSIBMADM	锁等待的信息
LOG_UTILIZATION	SYSIBMADM	日志利用率的信息
LONG_RUNNING_SQL	SYSIBMADM	执行时间最长的 SQL
SNAPAGENT_MEMORY_POOL SNAP_GET_AGENT_MEMORY_POOL	SYSIBMADM	代理级别的内存使用情况
SNAPBP MON_BP_UTILIZATION	SYSIBMADM	缓冲池的基本信息
SNAPDYN_SQL	SYSIBMADM	数据库中动态 SQL 的执行情况
SNAPLOCKWAIT MON_LOCKWAITS	SYSIBMADM	锁等待的信息
SNAPSTMT SNAP_GET_STMT	SYSIBMADM	应用中 SQL 语句的执行情况
SNAPTAB	SYSIBMADM	表的信息
SNAPTAB_REORG SNAP_GET_TAB_REORG	SYSIBMADM	重组信息
SNAPTBSP MON_TBSP_UTILIZATION	SYSIBMADM	表空间信息
TBSP_UTILIZATION	SYSIBMADM	表空间的利用情况
TOP_DYNAMIC_SQL MON_CURRENT_SQL	SYSIBMADM	消耗资源最多的 SQL 语句信息
MON_PKG_CACHE_SUMMARY	SYSIBMADM	PACKAGE 命中率信息
MON_CURRENT_UOW	SYSIBMADM	当前 UOW 事务信息
MON_WORKLOAD_SUMMARY	SYSIBMADM	WORKLOAD 信息汇总

(续表)

视　图　名	模　式　名	描　　述
MON_CONNECTION_SUMMARY	SYSIBMADM	数据库联系信息
MON_DB_SUMMARY	SYSIBMADM	数据库信息

表 3-2 列出了一些主要的视图，这些视图以较好的方式将快照获得信息进行良好的组织，使获得一些性能的信息变得简单容易。下面我们举一些示例。

3.4.1　监控缓冲池命中率

我们利用图 3-1 中的 SQL 语句来监控缓冲池的命中率。

```
select substr(bp_name,1,30) as bp_name ,
data_hit_ratio_percent,
index_hit_ratio_percent ,
total_hit_ratio_percent
from sysibmadm.bp_hitratio
 where bp_name not like 'IBMSYSTEM%' ;
```

把数据库内部 4 个默认创建的隐含缓冲池排除

```
BP_NAME          DATA_HIT_RATIO_PERCENT INDEX_HIT_RATIO_PERCENT TOTAL_HIT_RATIO_PERCENT
---------------- ---------------------- ----------------------- -----------------------
IBMDEFAULTBP                      71.11                   51.35                   70.88
CLPBUFFL                          61.84                    0.00                   60.99
CLPBUFFS                         -12.22                   13.33                  -12.17
```

如果缓冲池的物理读大于逻辑读，那么这个值是负值

图 3-1　监控缓冲池命中率

使用如下语句查看当前数据库缓冲池的同步和异步读写情况：

```
    select substr(bp name,1,20) as bp name, int ((1- (decimal(pool data p reads)
/ nullif(pool_data_l_reads,0)))*100) as data_hit_ratio, int ((1-(decimal(pool_
index p reads)/nullif(pool index l reads,0)))*100) as index hit ratio, int
((1-(decimal(pool data p reads+pool index p reads)/nullif((pool data l read
s+pool_index_l_reads),0)))*100) as BP_hit_ratio, int ((1-(decimal(pool_async_
data reads+pool async index reads)/nullif((pool async data reads+pool async
_index_reads+direct_reads),0)))*100) as Async_read_pct, int ((1-(decimal
(direct writes)/nullif(direct reads,0)))*100) as Direct RW Ratio from TABLE
(MON_GET_BUFFERPOOL('',-2)) as snapshot_bp where bp_name not like 'IBMSYSTEM%';
```

输出信息如下：

```
BP NAME      DATA HIT RATIO  INDEX HIT RATIO BP HIT RATIO ASYNC READ PCT DIRECT RW RATIO
----------- ----------- ------------- --------- ------------ -----------
IBMDEFAULTBP          99            99          99            99            99
BP_32K                97            98          97             -             -
```

3.4.2 监控 Package Cache 大小

可以利用如下 SQL 语句来监控应用程序包的大小，其中 PKG_CACHE_LOOKUPS 表示执行计划在程序包中直接从 Package Cache 中查找的次数，PKG_CACHE_INSERTS 表示由于应用程序包缓存中没有存在 SQL 语句的执行计划而需要重新插入的次数。

```
  select DECIMAL(1 - ( PKG CACHE INSERTS *1.0 / PKG CACHE LOOKUPS ), 3, 2)
         as PKG CACHE HIT, PKG CACHE NUM OVERFLOWS,
         PKG CACHE SIZE TOP
  from SYSIBMADM.SNAPDB;
```

输出如下：

```
PKG CACHE HIT PKG CACHE NUM OVERFLOWS PKG CACHE SIZE TOP
------------- ----------------------- --------------------
       0.89                       0            1313665

  1 record(s) selected.
```

3.4.3 监控执行成本最高的 SQL 语句

可以使用性能管理视图来监控高成本应用程序。snapappl 视图可以帮助我们监控可能正在执行大型表扫描操作的应用程序：

```
    select agent id,rows selected,rows read from sysibmadm.snapappl order by
rows_read desc fetch first 10 rows only;
```

rows_selected 表示返回给应用程序的行数，rows_read 表示在基本表中访问的行数。如果选择率很低，那么表示该应用程序可能正在执行表扫描操作，通过创建索引可以避免应用程序执行该操作。使用此视图来标识可能有问题的查询，然后可以进行进一步调查，查看 SQL 执行计划以确定是否能够减少查询在执行时读取的行数。

3.4.4　监控运行时间最长的 SQL 语句

我们可以使用 LONG_RUNNING_SQL 管理视图来监控当前正在执行的运行时间最长的查询，如图 3-2 所示。

```
select substr(appl_name,1,15) as Appl_name ,
elapsed_time_min as "Elapsed Min." ,
appl_status as "Status ",
substr(authid,1,10) as auth_id ,
substr(inbound_comm_address,1,15)
   as "IP Address",
substr(stmt_text,1,30) as "SQL Statement"
from sysibmadm.long_running_sql
order by 2 desc ;

APPL_NAME       Elapsed Min. Status                 AUTH_ID
--------------- ------------ ---------------------- --------
db2taskd                   - CONNECTED              INST461
db2stmm                    - CONNECTED              INST461
db2bp                     16 LOCKWAIT               INST461
db2bp                      0 UOWWAIT                INST461
db2bp                      0 UOWEXEC                INST461

IP Address      SQL Statement
--------------- ------------------------------
-               -
-               -
*LOCAL.inst461  update clpm.hist1 set balance=
*LOCAL.inst461  update clpm.hist1 set balance=
*LOCAL.inst461  select substr(appl_name,1,15)
```

图 3-2　使用 LONG_RUNNING_SQL 监控运行时间最长的 SQL 语句

通过使用此视图，我们可以监控那些查询已运行的时间长度以及这些查询的状态。如果某个查询已执行了很长时间并且正在等待锁，我们还可以使用对特定代理程序标识执行查询的 LOCKWAITS 或 LOCK_HELD 管理视图来进行进一步调查。LONG_RUNNING_SQL 视图还会指出正在执行的语句并允许标识可能有问题的 SQL。

在 DB2 V10.5 中，建议使用 MON_CURRENT_SQL 来替代 LONG_RUNNING_SQL。在 MON_CURRENT_SQL 中提供了更多的正在运行的 SQL 的信息，包括 ROWS_READ、ROWS_RETURNED 等。

3.4.5　监控 SQL 准备和预编译时间最长的 SQL 语句

可以使用 QUERY_PREP_COST 来对已确定有问题的查询进行故障诊断。此视图可以指出查询的运行频率以及这些查询中每个查询的平均执行时间，如图 3-3 所示。PREP_TIME_PERCENT 向我们指出准备查询时耗用的时间在查询执行时间中所占的百分比。如果编译和优化查询时耗用的时间几乎与查询的执行时间一样长，那么可以降低优化级别使该查询更快地完成优化，从而更快地返回结果。但是，如果某个查询需要相当长的时间来进行准备，但要执行数千次(而不必再次进行准备)，那么更改优化并不能提高查询

性能。

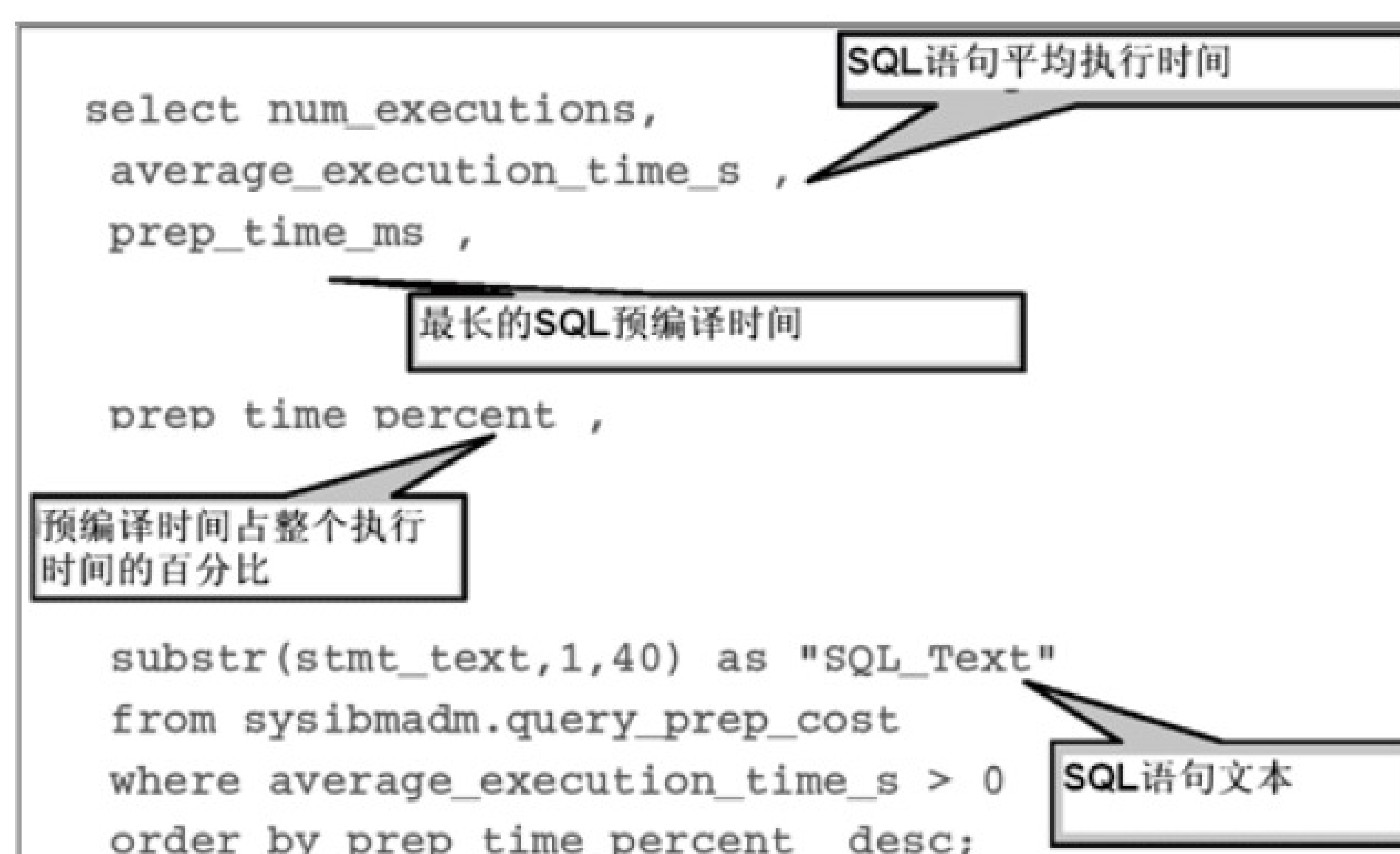


图 3-3　使用 QUERY_PREP_COST 监控 SQL 准备和预编译时间

3.4.6　监控执行次数最多的 SQL 语句

可以使用 TOP_DYNAMIC_SQL 视图来标识执行次数最多、运行时间最长和排序次数最多的动态 SQL 语句，如图 3-4 所示。有了此信息，我们在进行 SQL 调整工作时就可以把注意力放在代表某些最大资源使用者的查询上。

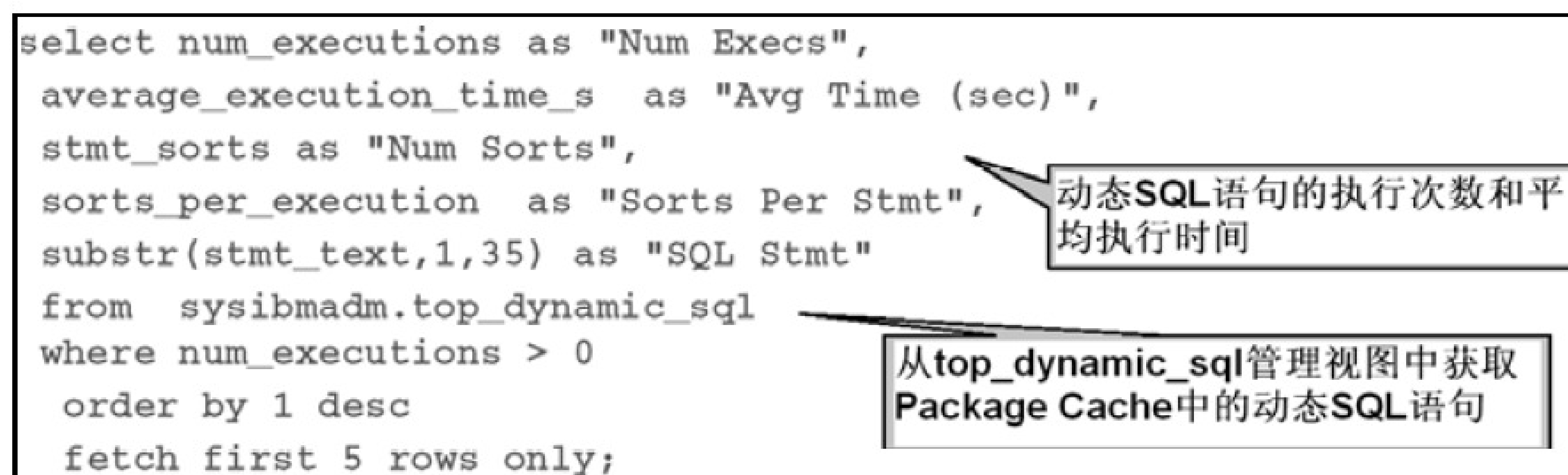


图 3-4　标识运行频率最高的动态 SQL 语句

图 3-4 中的语句返回执行频率最高的 5 条动态 SQL 语句的所有执行时间、排序执行次数和语句文本详细信息，如图 3-5 所示。

为了标识执行次数最多的动态 SQL 语句，请检查 AVERAGE_EXECUTION_TIME_S 值最大的 5 个查询。

```
select num_executions as "Num Execs",
 average_execution_time_s  as "Avg Time (sec)",
 stmt_sorts as "Num Sorts",
 sorts_per_execution  as "Sorts Per Stmt",
 substr(stmt_text,1,35) as "SQL Stmt"
 from  sysibmadm.top_dynamic_sql
 where num_executions > 0
      order by 2 desc
   fetch first 5 rows only;
```

该SQL返回平均执行时间最长的SQL语句，并且同时返回该SQL的执行次数、排序次数和排序时间

图 3-5　返回结果

3.4.7　监控排序次数最多的 SQL 语句

为了查看排序次数最多的动态 SQL 语句的详细信息，发出以下语句：

```
    select STMT_SORTS, SORTS_PER_EXECUTION, substr(STMT_TEXT,1,60) as STMT_TEXT
from SYSIBMADM.TOP_DYNAMIC_SQL
    order by STMT_SORTS desc fetch first 5 rows only;
```

3.4.8　监控锁等待时间

可以通过图 3-6 所示的 SQL 语句监控锁等待时间，定位是否存在严重的锁等待。

应用名称和AUTH_ID可能很长，为了便于显示，可以使用substr截取部分信息

```
select
 substr(ai.appl_name,1,20) as appl_name ,
 substr(ai.primary_auth_id,1,10) as auth_id ,
 ap.lock_waits as lock_waits,
 ap.lock_wait_time / 1000 as "Total Wait (s)",
 (ap.lock_wait_time / ap.lock_waits ) as "Avg Wait (ms)"

 from sysibmadm.snapappl_info ai, sysibmadm.snapappl ap
 where ai.agent_id = ap.agent_id
 and ap.lock_waits > 0 ;
```

snapappl_info和snapappl两张视图中存放关于应用程序相关的详细监控信息

```
APPL_NAME            AUTH_ID    LOCK_WAITS           Total Wait (s)       Avg Wait (ms)
-------------------- ---------- -------------------- -------------------- --------------------
db2bp                INST461                       1                  162               162168
```

图 3-6　监控锁等待时间

3.4.9　监控 Lock Chain

有时锁等待会造成级联循环，形成锁循环链(Lock Chain)，图 3-7 所示的这条 SQL 语句可以定位持有锁和等待锁的应用。

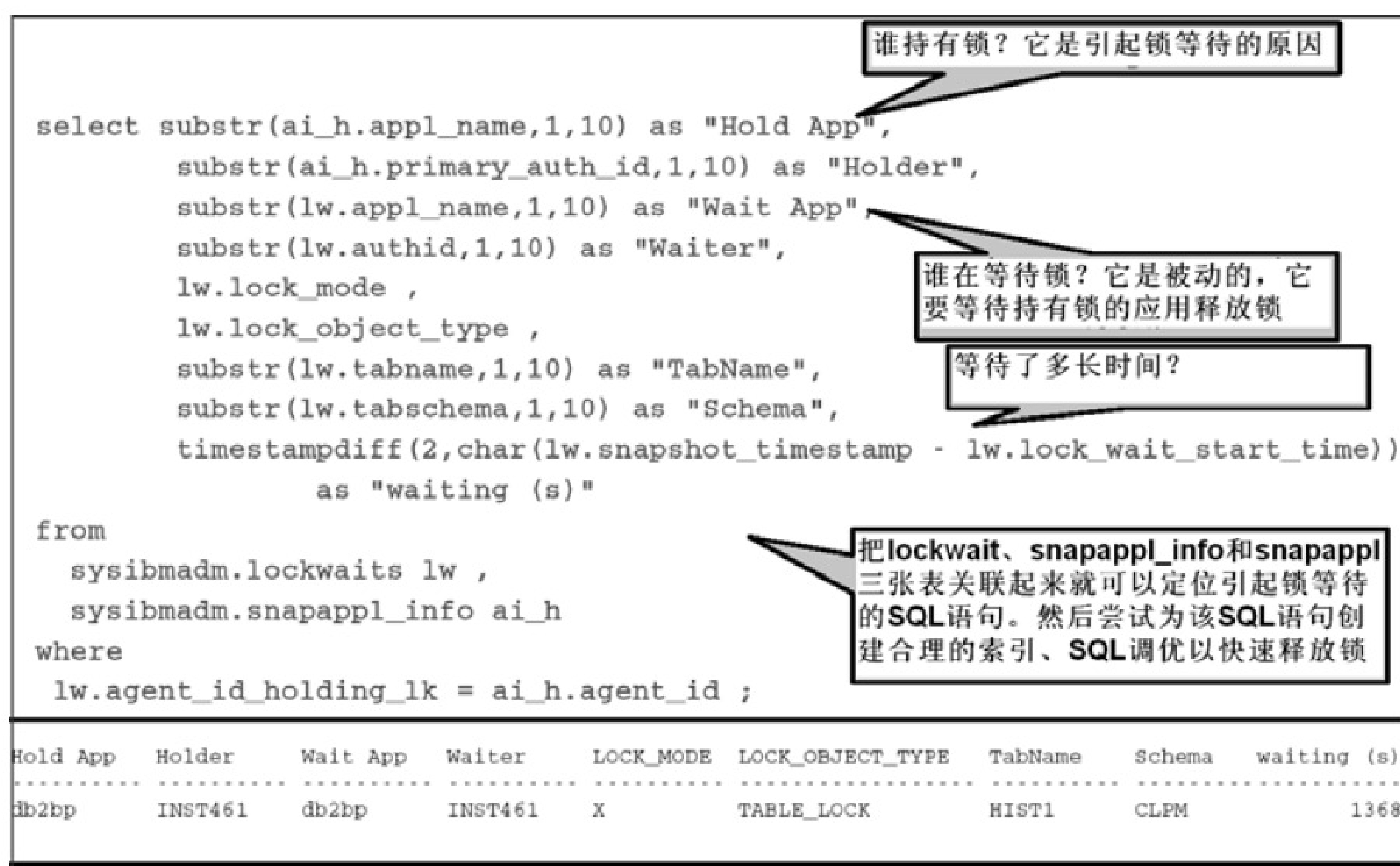


图 3-7　监控 Lock Chain

另外还可以通过脚本来抓取锁等待和锁链信息，我们可以把脚本部署到监控系统中，在夜间业务系统发生锁等待时，可以使用脚本抓取信息。

如下是抓取锁等待信息的脚本，供大家参考：

```
    #/bin/sh
    # Creation Date:      2011.06.27
    # Version: 1.0
    # Modification History
    #
    db locks(){
       currTime='date'
       echo "Current time: ${currTime}"
       db=$1
       db2pd -d $db -wlock>$logdir/wlock.list
     echo " "
     echo "**********************"
     echo "Current LockWait list"
     echo"****************************************************************"
     cat $logdir/wlock.list
     echo"****************************************************************"

     cat $logdir/wlock.list|grep -v Locks|grep -i G|awk '{print $1}'
>$logdir/holder
```

```
    cat $logdir/wlock.list|grep -v G|grep -v Locks|grep -i w |awk '{print $1}'
>$logdir/waiter

    #holder collect info
      echo "Collect the info for Holder Agent"
    echo "*****************************************************************"
      id=0
      num='cat $logdir/holder|wc -l'
      while [ "$id" -lt "$num" ]
      do
      id='expr $id + 1'
      agentid='head -$id $logdir/holder |tail -1| awk '{print $1}''
      db2pd -d $db -app $agentid > $agentid.txt
      CA='grep $agentid $agentid.txt | sed -n '1p' | awk '{ print $7}''
      CS='grep $agentid $agentid.txt | sed -n '1p' | awk '{ print $8}''
      LA='grep $agentid $agentid.txt | sed -n '1p' | awk '{ print $9}''
      LS='grep $agentid $agentid.txt | sed -n '1p' | awk '{ print $10}''
    echo "Collect the dynamic sql to $agentid"
    if [ ${CA} -eq 0 -a ${CS} -eq 0 ]
      then
      echo "db2pd -d $db -dyn |awk 'flag==1 && /^0x/ {flag=0} flag==1{print}
\$2==$LA && \$3==$LS  {print \$0;flag=1;}'" > 1.tmp
    echo "-----------------------------------------------------------------"
      sh 1.tmp
      rm 1.tmp
      else
      echo "db2pd -d $db -dyn |awk 'flag==1 && /^0x/ {flag=0} flag==1{print}
\$2==$CA && \$3==$CS  {print \$0;flag=1;}'" >2.tmp
    echo "-----------------------------------------------------------------"
      sh 2.tmp
      rm 2.tmp
      fi
      #getdyn $db $line |tee -a $logdir/hold.$line.$da
      done
    echo "*****************************************************************"

    #waiter collect info
      echo "Collect the info for Waiter Agent"
    echo "******************************************************************"
      id=0
      num='cat $logdir/waiter|wc -l'
      while [ "$id" -lt "$num" ]
      do
      id='expr $id + 1'
```

```
        agentid='head -$id $logdir/waiter |tail -1| awk '{print $1}''
        db2pd -d $db -app $agentid > $agentid.txt
        CA='grep $agentid $agentid.txt | sed -n '1p' | awk '{ print $7}''
        CS='grep $agentid $agentid.txt | sed -n '1p' | awk '{ print $8}''
        LA='grep $agentid $agentid.txt | sed -n '1p' | awk '{ print $9}''
        LS='grep $agentid $agentid.txt | sed -n '1p' | awk '{ print $10}''
     echo "Collect the dynamic sql to $agentid"
     if [ ${CA} -eq 0 -a ${CS} -eq 0 ]
        then
        echo "db2pd -d $db -dyn |awk 'flag==1 && /^0x/ {flag=0} flag==1{print}
\$2==$LA && \$3==$LS  {print \$0;flag=1;}'" > 1.tmp
     echo "------------------------------------------------------------------"
        sh 1.tmp
        rm 1.tmp
        else
        echo "db2pd -d $db -dyn |awk 'flag==1 && /^0x/ {flag=0} flag==1{print}
\$2==$CA && \$3==$CS  {print \$0;flag=1;}'" >2.tmp
     echo "-------------------------------------------------------------------"
        sh 2.tmp
        rm 2.tmp
        fi
        #getdyn $line |tee -a $logdir/wait.$line.$da
        done
     echo "******************************************************************"

    }
    . ${HOME}/.profile
    today='date +%m%d%M%H%S'
    logdir=/tmp
    logfile=${logdir}/db locks $1 ${today}.log
    echo "##############################################" |tee>${logfile}
    echo "#        DB2 LOCKWAIT CHECK START..."            |tee>>${logfile}
    echo "##############################################" |tee>>${logfile}
    db locks $1 |tee>>${logfile}
    echo "##############################################" |tee>>${logfile}
    echo "#        DB2 LOCKWAIT CHECK END..."              |tee>>${logfile}
    echo "##############################################" |tee>>${logfile}
    cat ${logfile}
```

3.4.10 监控锁内存的使用

可以用图 3-8 所示的 SQL 语句判断锁内存的使用。

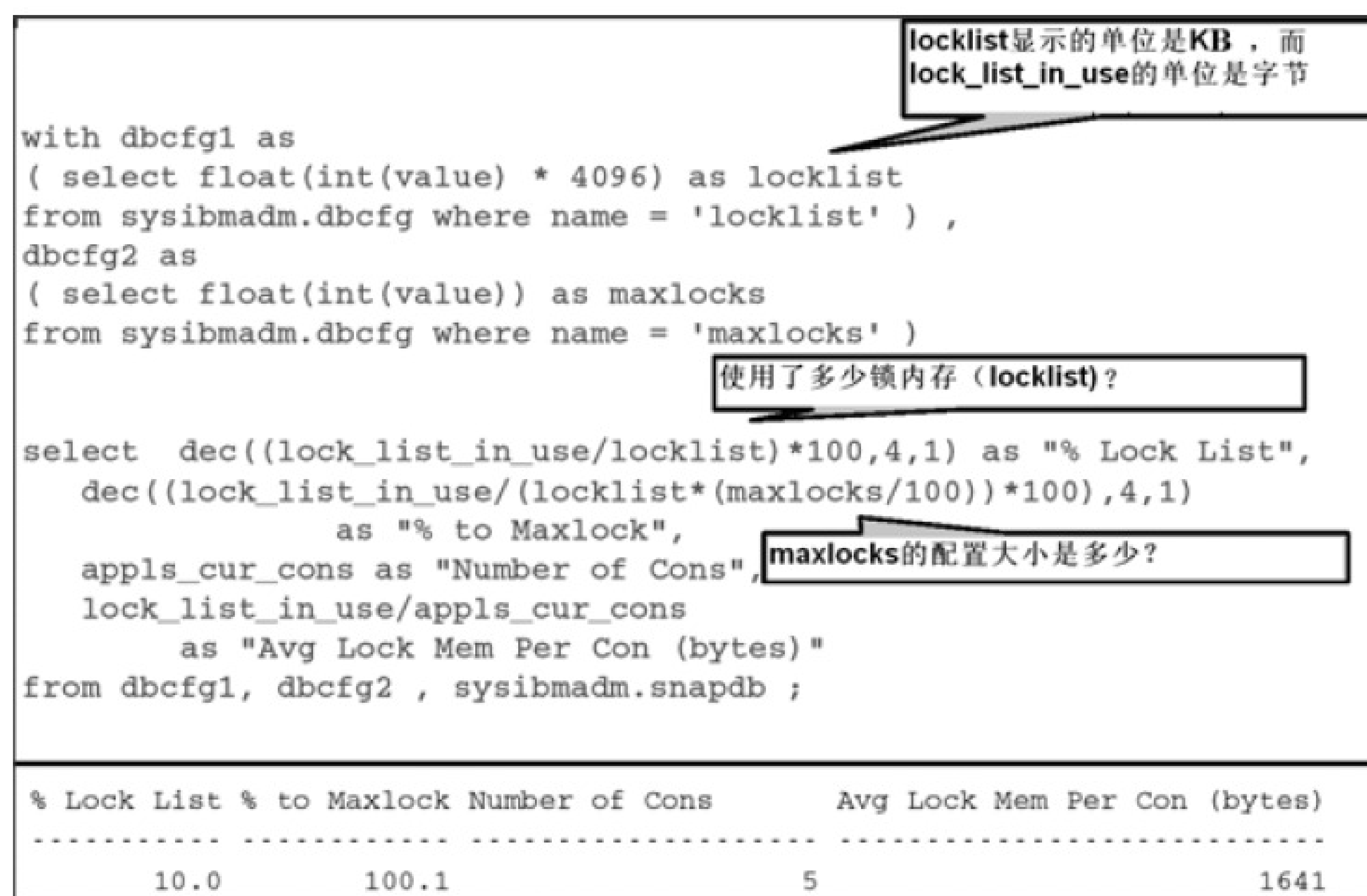


图 3-8　监控锁内存的使用情况

3.4.11　监控锁升级、死锁和锁超时

图 3-9 所示的 SQL 语句可以监控锁升级、死锁和锁超时。



图 3-9　监视锁升级、死锁和锁超时

我们还可以通过如下语句来确定某个表上有哪些锁，并列出这些锁的具体信息：

```
    SELECT trim(substr(A.TABSCHEMA,1,8))||'.'||substr(A.TABNAME, 1,15) as
TABNAME, A.LOCK_MODE,A.DBPARTITIONNUM,A.AGENT_ID, SUBSTR(B.APPL_ID,1,10) AS
APPL_ID, B.CLIENT_PID,SUBSTR(CLIENT_PLATFORM,1,8) AS CPLATFORM,
```

```
SUBSTR(B.CLIENT NNAME,1,8) AS CLIENT NAME FROM SYSIBMADM.SNAPLOCK A,
SYSIBMADM.APPLICATIONS B WHERE A.AGENT_ID = B.AGENT_ID AND TABNAME='tabname';
```

输出信息如下：

```
TABNAME  LOCK_MODE     DBPARTITIONNUM AGENT_ID    APPL_ID  CLIENT_PID CPLATFORM CLIENT_NAME
-----------------  ------------- -------- ---------- ----------- --------- ----------
SAPTOOLS.SCSTATS_SAP_WLM          IN        0  502 *LOCAL.DB2 8192132  AIX64   BPMPPDDA
```

3.4.12　监控全表扫描的 SQL

图 3-10 所示的 SQL 语句监控数据库中哪些表上采用的是全表扫描方式。

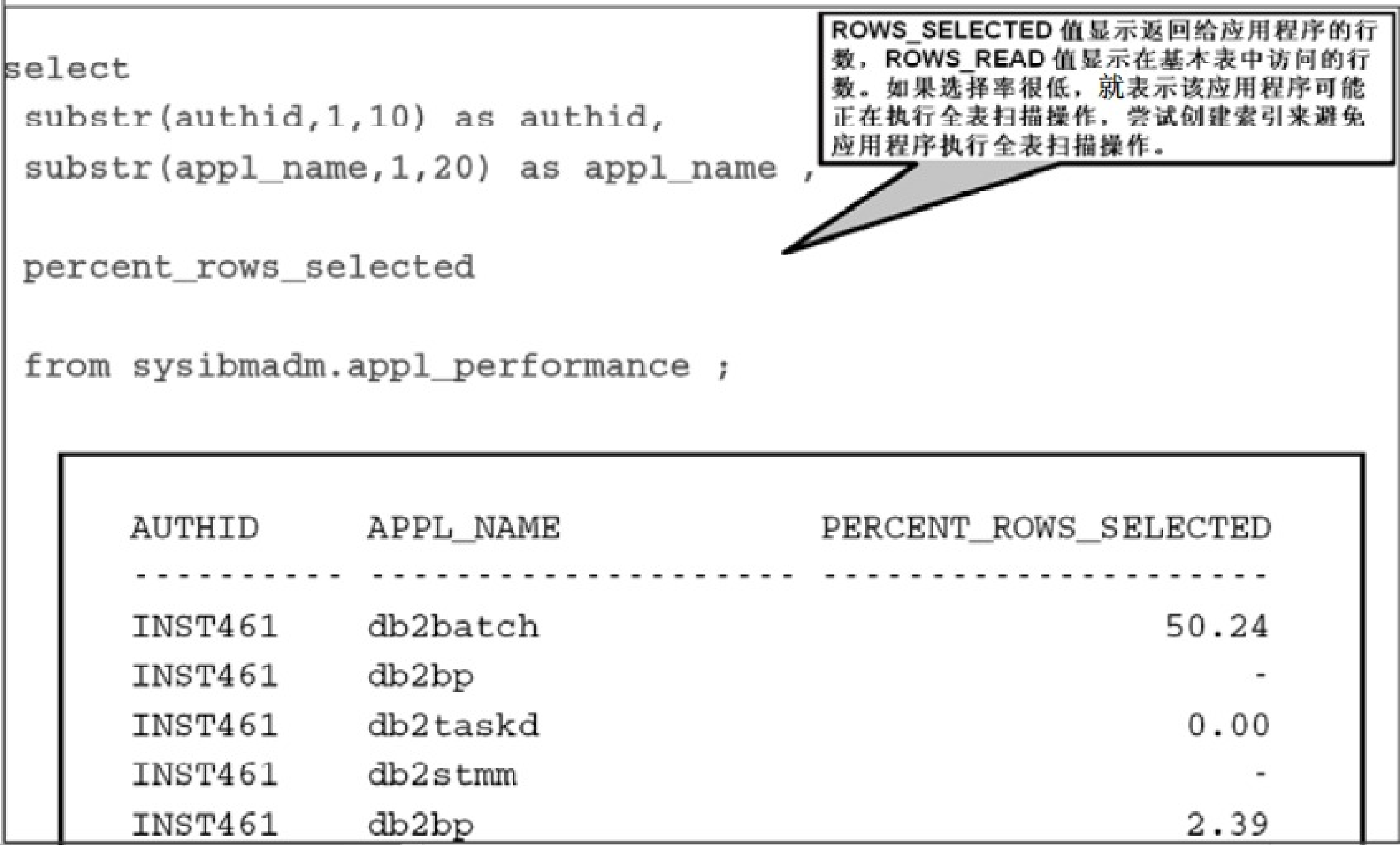


图 3-10　监控进行全表扫描的 SQL

除了使用图 3-10 所示的 SQL 语句来监控全表扫描的表，我们还可以使用 db2pd 工具来监控在哪些表上进行了全表扫描及扫描的次数。我们在 3.5 节中有这样的案例。

3.4.13　检查页清理器是否足够

页清理器在以下 3 种情况下触发：

- 缓冲池中的脏页数量达到了更改页阀值(CHNGPGS_THRESH)。
- 日志的更改到达了 softmax 软检查点。

- 当脏页被选择作为牺牲页而释放 bufferpool 空间时，页清理器要把脏页写到硬盘。

图 3-11 所示的 SQL 语句可以监控数据库中在上述 3 种情况下页清理器所占的百分比，以此来判断是否配置了足够的页清理器(num_iocleaners)。

注意：

如果把 DB2 注册变量 DB2_USE_ALTERNATE_PAGE_CLEANING 设置为 ON 的话(页清除程序总是处于激活状态，不用等待条件值触发)，图 3-11 中的这条 SQL 语句将无效。

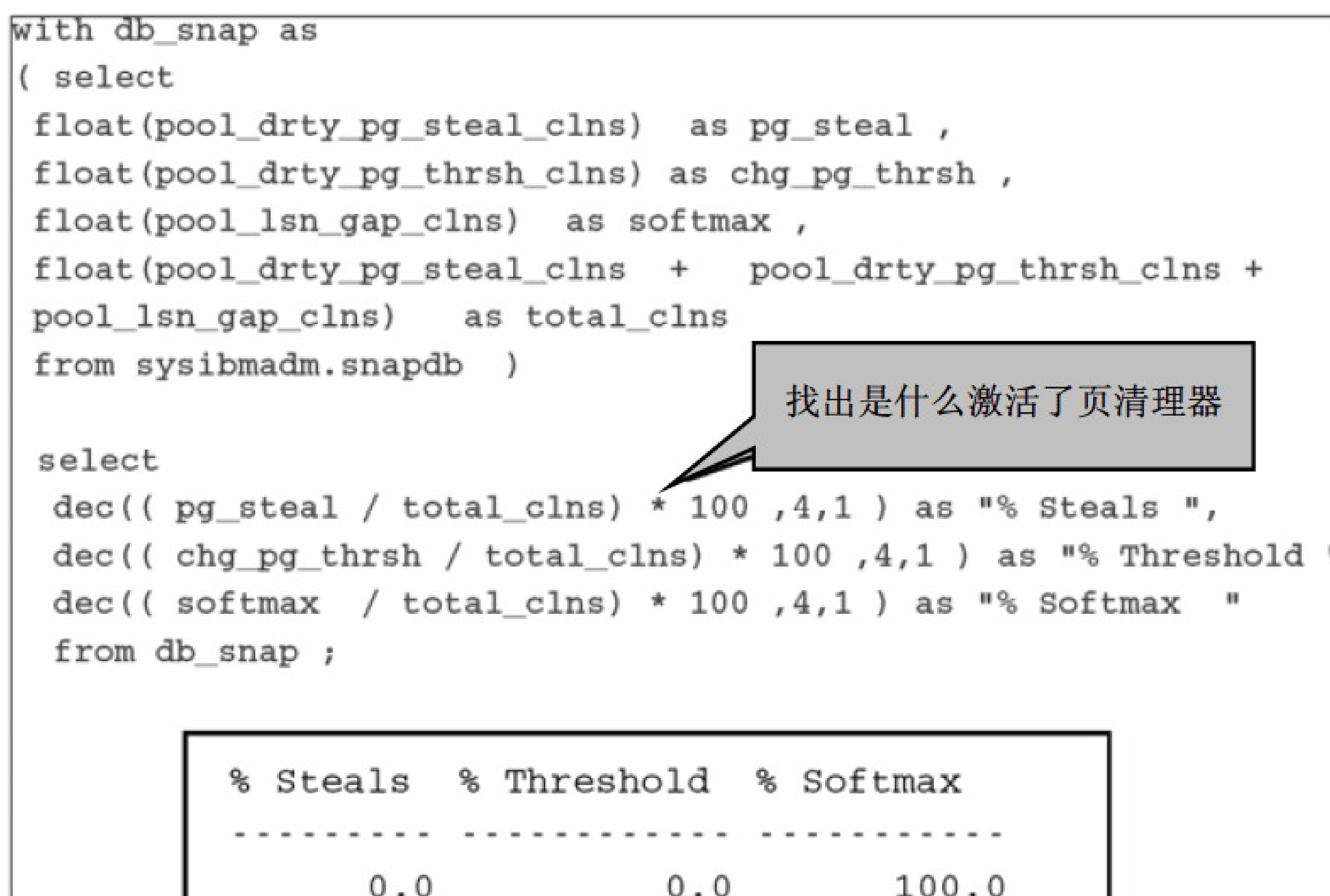
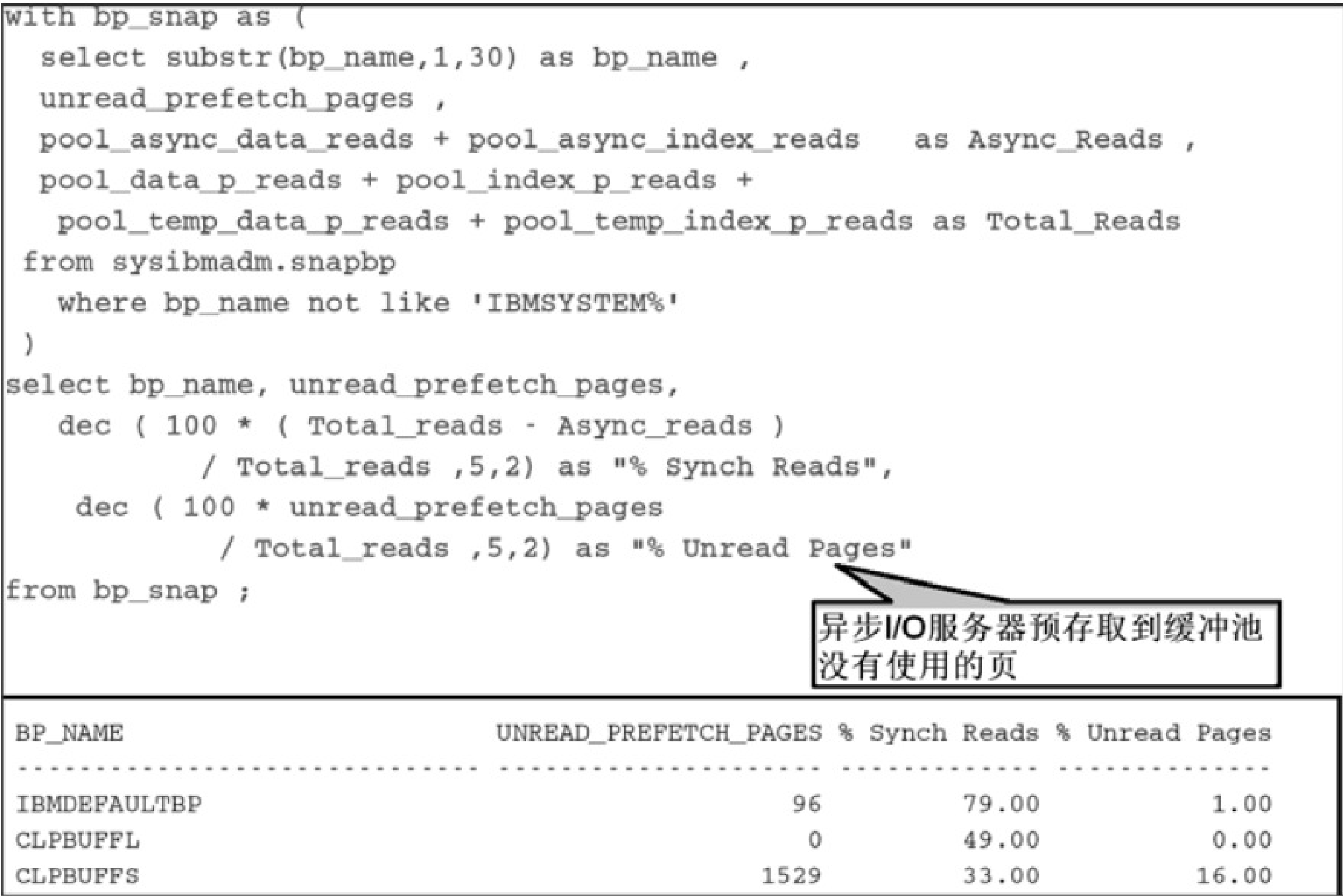


图 3-11　检查页清理器是否足够

3.4.14　监控 prefecher 是否足够

图 3-12 所示的 SQL 语句可以监控数据库中是否配置了足够的异步 I/O 服务器个数(num_ioservers)。



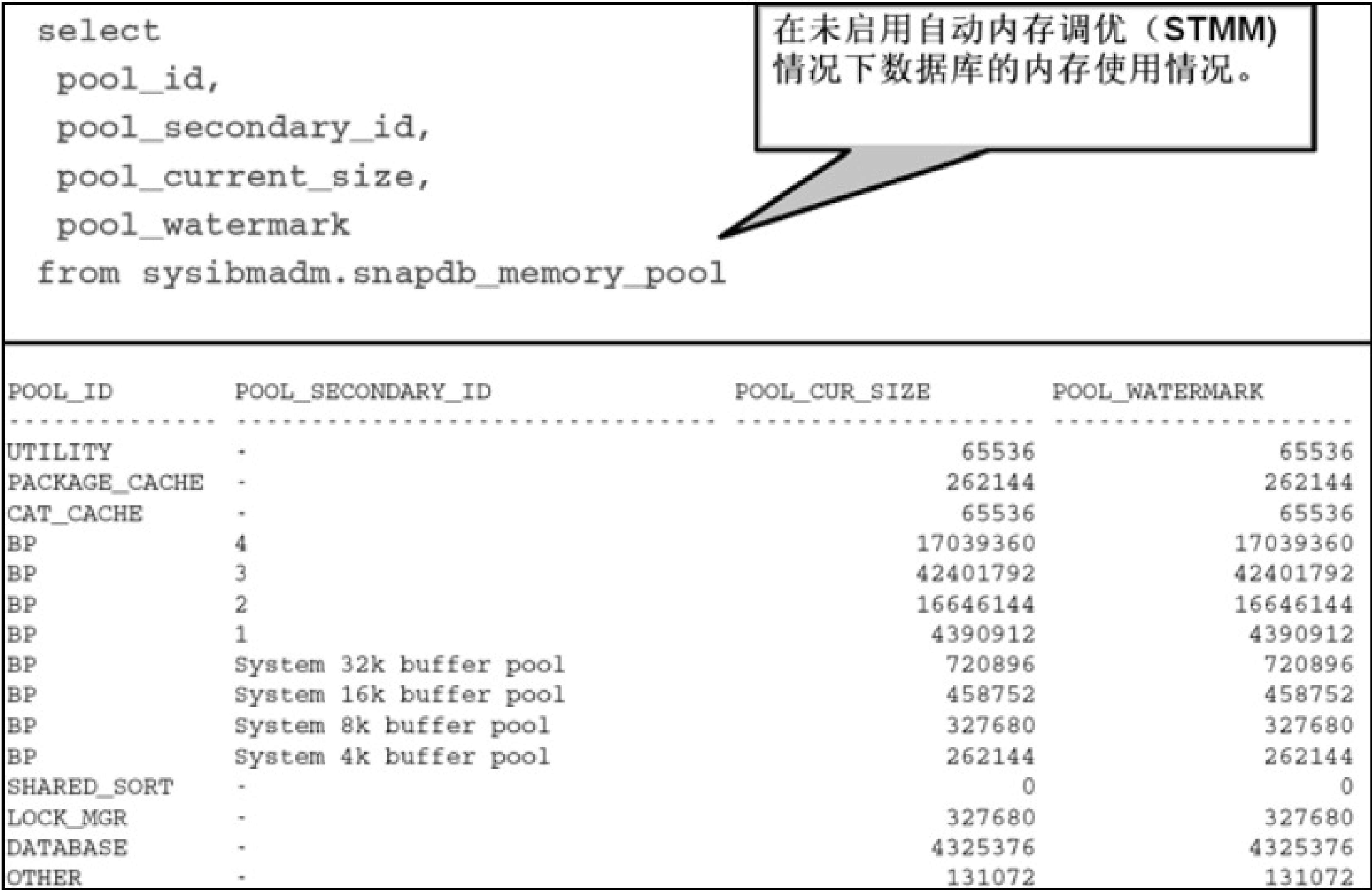
```
with bp_snap as (
  select substr(bp_name,1,30) as bp_name ,
  unread_prefetch_pages ,
  pool_async_data_reads + pool_async_index_reads   as Async_Reads ,
  pool_data_p_reads + pool_index_p_reads +
   pool_temp_data_p_reads + pool_temp_index_p_reads as Total_Reads
 from sysibmadm.snapbp
   where bp_name not like 'IBMSYSTEM%'
 )
select bp_name, unread_prefetch_pages,
   dec ( 100 * ( Total_reads - Async_reads )
           / Total_reads ,5,2) as "% Synch Reads",
    dec ( 100 * unread_prefetch_pages
            / Total_reads ,5,2) as "% Unread Pages"
from bp_snap ;
```


BP_NAME	UNREAD_PREFETCH_PAGES	% Synch Reads	% Unread Pages
IBMDEFAULTBP	96	79.00	1.00
CLPBUFFL	0	49.00	0.00
CLPBUFFS	1529	33.00	16.00

图 3-12　监控 prefecher 是否足够

3.4.15　监控数据库内存使用

图 3-13 所示的 SQL 语句可以监控数据库的内存使用情况。



```
select
 pool_id,
 pool_secondary_id,
 pool_current_size,
 pool_watermark
from sysibmadm.snapdb_memory_pool
```


POOL_ID	POOL_SECONDARY_ID	POOL_CUR_SIZE	POOL_WATERMARK
UTILITY	-	65536	65536
PACKAGE_CACHE	-	262144	262144
CAT_CACHE	-	65536	65536
BP	4	17039360	17039360
BP	3	42401792	42401792
BP	2	16646144	16646144
BP	1	4390912	4390912
BP	System 32k buffer pool	720896	720896
BP	System 16k buffer pool	458752	458752
BP	System 8k buffer pool	327680	327680
BP	System 4k buffer pool	262144	262144
SHARED_SORT	-	0	0
LOCK_MGR	-	327680	327680
DATABASE	-	4325376	4325376
OTHER	-	131072	131072

图 3-13　监控数据库内存使用情况

3.4.16　监控日志使用情况

图 3-14 所示的 SQL 语句可以监控数据库日志空间使用情况。



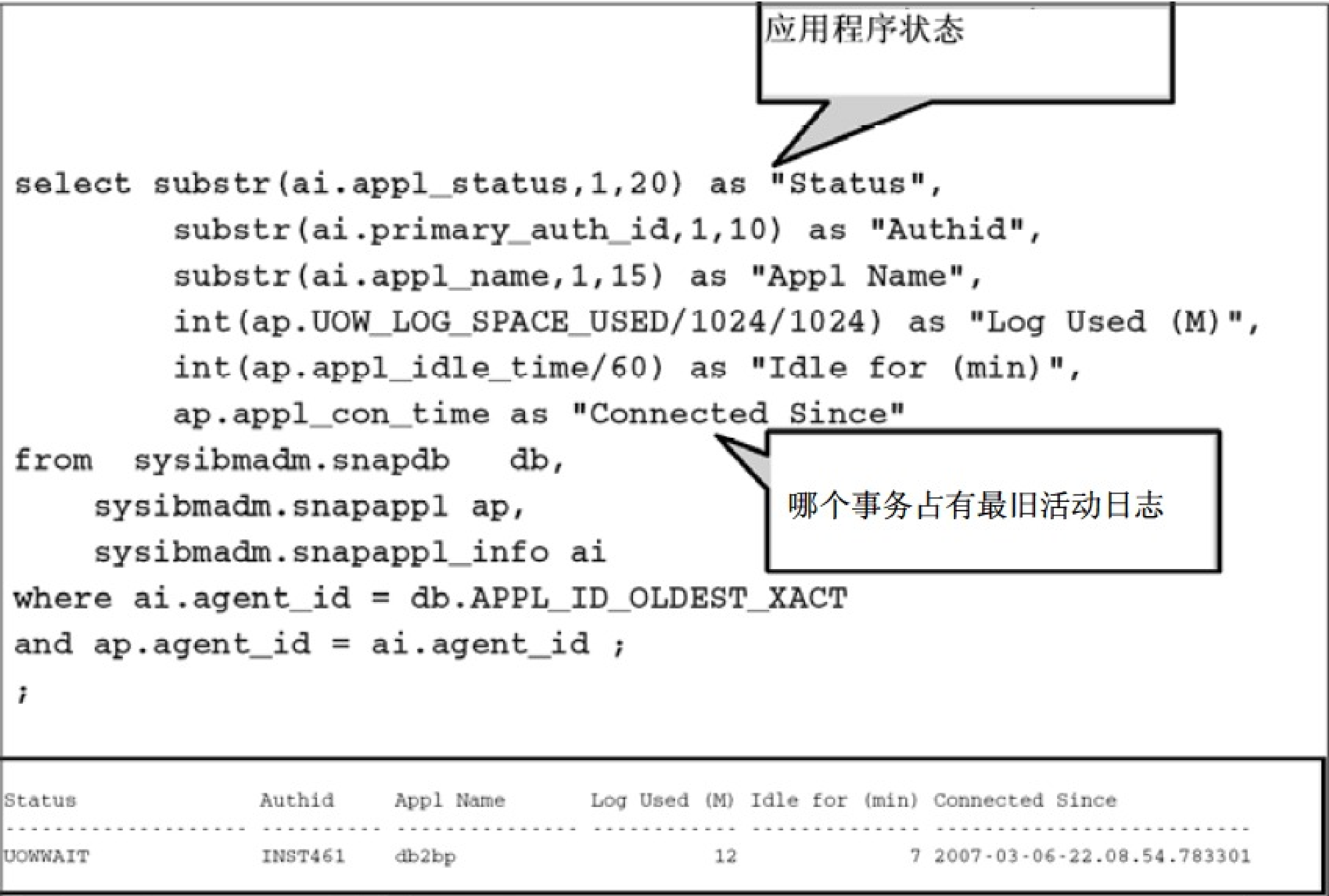

```
select
        int(total_log_used/1024/1024) as "Log Used (Meg)",
        int(total_log_available/1024/1024)
                as "Log Space Free (Meg)",
        int((float(total_log_used) /
                float(total_log_used+total_log_available))*100)
                as "Pct Used",
        int(tot_log_used_top/1024/1024) as "Max Log Used (Meg)",
        int(sec_log_used_top/1024/1024) as "Max Sec. Used (Meg)",
        int(sec_logs_allocated) as "Secondaries"
from sysibmadm.snapdb ;
```

```
Log Used(Meg) Log Space Free(Meg) Pct Used   Max Log Used (Meg) Max Sec. Used (Meg) Secondaries
---------     ------------------- ---------------------------- ------------------ -----------
       12                       3         76                 12                 10          14
```

图 3-14　监控数据日志空间的使用情况

3.4.17　监控占用日志空间最旧的事务

我们在数据库中经常碰到 SQL0964C 报错，通常这个错误表示日志空间满，可以尝试分配更多的日志空间。但造成 SQL0964C 还有另外一种原因：在日志空间并未用尽的情况下，当某个占有最旧活动日志的应用长时间未做提交操作时，会阻止日志的 LSN 的分配，造成日志空间无法使用，这同样会引发这一日志满的报错。对于这种情况，可以提交该事务或利用 FORCE 命令终止此应用程序，以便释放它所占用的日志空间，使 LSN 可以继续分配，空闲的日志空间可用。图 3-15 所示的 SQL 语句可以监控数据库中哪个事务持有最旧的活动日志。




```
select substr(ai.appl_status,1,20) as "Status",
       substr(ai.primary_auth_id,1,10) as "Authid",
       substr(ai.appl_name,1,15) as "Appl Name",
       int(ap.UOW_LOG_SPACE_USED/1024/1024) as "Log Used (M)",
       int(ap.appl_idle_time/60) as "Idle for (min)",
       ap.appl_con_time as "Connected Since"
from  sysibmadm.snapdb   db,
    sysibmadm.snapappl ap,
    sysibmadm.snapappl_info ai
where ai.agent_id = db.APPL_ID_OLDEST_XACT
and ap.agent_id = ai.agent_id ;
;
```

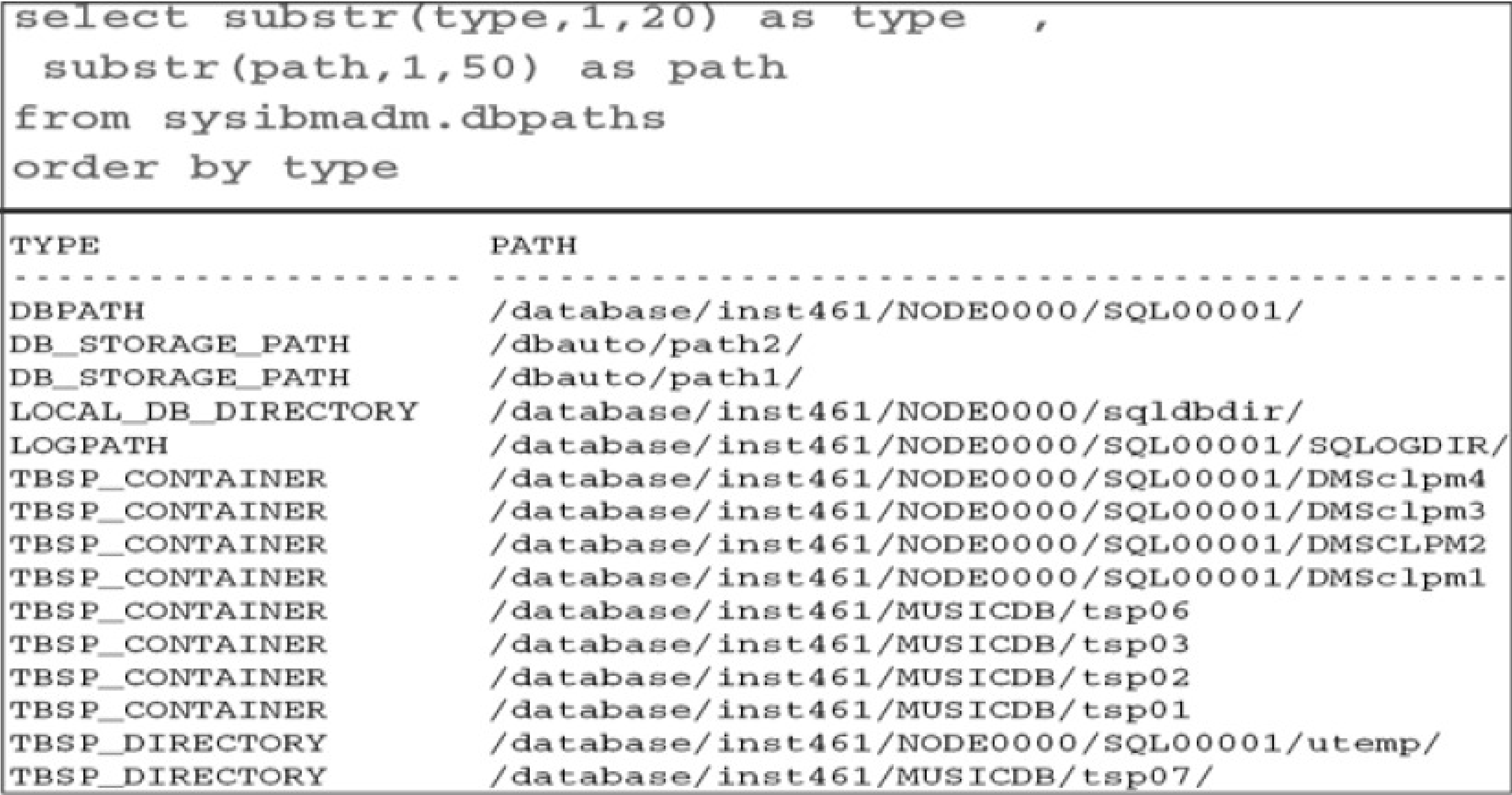
```
Status               Authid     Appl Name       Log Used (M) Idle for (min) Connected Since
-------------------- ---------- --------------- ------------ -------------- --------------------------
UOWWAIT              INST461    db2bp                     12              7 2007-03-06-22.08.54.783301
```

图 3-15　监控占用最旧活动日志的事务

一旦定位这个事务，我们就可以终止这个事务，或者查看这个事务的详细信息，看这个事务正在执行的 SQL，从而帮助我们进一步定位问题所在。

3.4.18　监控存储路径

图 3-16 所示的 SQL 语句可以监控数据库表空间中的容器存储路径。



```
select substr(type,1,20) as type   ,
 substr(path,1,50) as path
from sysibmadm.dbpaths
order by type
```

```
TYPE                 PATH
-------------------- --------------------------------------------------
DBPATH               /database/inst461/NODE0000/SQL00001/
DB_STORAGE_PATH      /dbauto/path2/
DB_STORAGE_PATH      /dbauto/path1/
LOCAL_DB_DIRECTORY   /database/inst461/NODE0000/sqldbdir/
LOGPATH              /database/inst461/NODE0000/SQL00001/SQLOGDIR/
TBSP_CONTAINER       /database/inst461/NODE0000/SQL00001/DMSclpm4
TBSP_CONTAINER       /database/inst461/NODE0000/SQL00001/DMSclpm3
TBSP_CONTAINER       /database/inst461/NODE0000/SQL00001/DMSCLPM2
TBSP_CONTAINER       /database/inst461/NODE0000/SQL00001/DMSclpm1
TBSP_CONTAINER       /database/inst461/MUSICDB/tsp06
TBSP_CONTAINER       /database/inst461/MUSICDB/tsp03
TBSP_CONTAINER       /database/inst461/MUSICDB/tsp02
TBSP_CONTAINER       /database/inst461/MUSICDB/tsp01
TBSP_DIRECTORY       /database/inst461/NODE0000/SQL00001/utemp/
TBSP_DIRECTORY       /database/inst461/MUSICDB/tsp07/
```

图 3-16　监控容器的存储路径

3.4.19　追踪监控历史

图 3-17 所示的 SQL 语句可以监控数据库的监控历史。

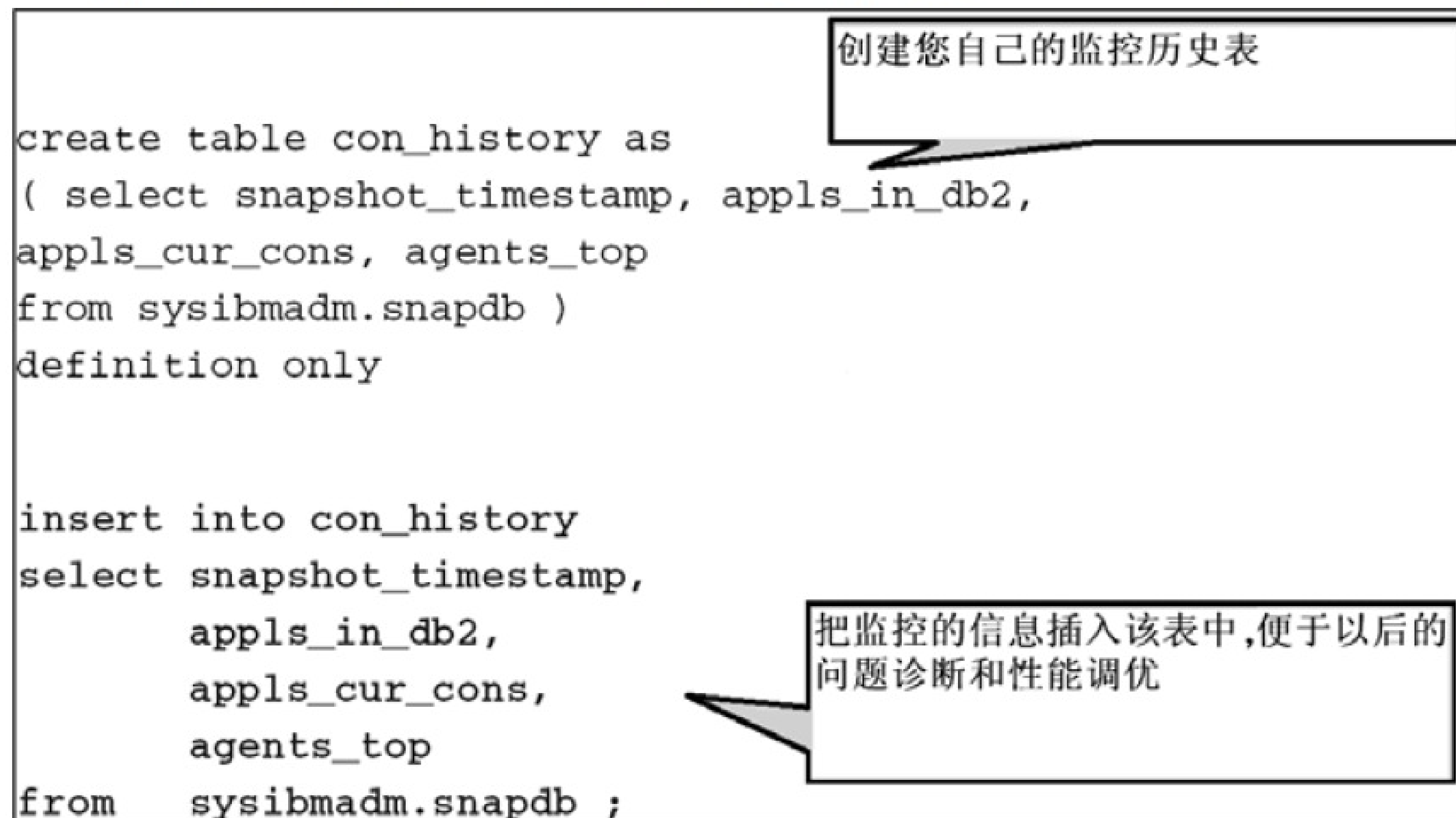


图 3-17　监控数据库的监控历史

3.5　db2pd

db2pd 是新的实用程序，可用来从正在运行的 DB2 实例或数据库检索统计信息，类似于 Informix 的 onstat 实用程序。该工具可以提供大量有用信息以帮助进行故障诊断和问题确定、性能提高和应用程序开发设计，这些信息包括：

- 操作系统、DBM 和 DB 配置参数
- 锁定
- 缓冲池
- 表空间
- 容器
- 动态 SQL 语句
- 代理进程
- 应用程序
- 内存池和内存集
- 事务

- 日志
- 其他

该工具不需要获得任何锁存器或使用任何引擎资源就可以收集这些信息。因此，当 db2pd 收集信息时，有可能(并且预计会这样)会检索到正在更改的信息；这样的话，数据可能不是十分准确。但是，在不加锁的情况下收集信息有两个好处：检索速度更快并且不会争用引擎资源。

3.5.1 常用 db2pd 监控选项和示例

下面我们举一些使用 db2pd 进行快速故障诊断的示例。

例 3-2 诊断造成锁等待的表上的加锁情况。

首先利用 db2pd –db musicdb –applications 命令，定位 status=Lock-Wait 的应用程序的句柄 AppHandl；然后再根据这个 AppHandl 定位事务句柄(TranHdl)，然后我们再根据这个事务句柄定位所在的表及表上的加锁情况，如图 3-18 和图 3-19 所示。

```
db2pd -db musicdb -applications

Database Partition 0 -- Database MUSICDB -- Active -- Up 0 days 00:05:13

Applications:
Address     AppHandl [nod-index] NumAgents  CoorTid   Status        Appid
0x0091FF30  21       [000-00021] 1          1404      Lock-wait     *LOCAL.INST45.050413230634
0x0091EC50  13       [000-00013] 1          2276      UOW-Waiting   *LOCAL.INST45.050413230250
0x0091EBF0  12       [000-00012] 1          3840      UOW-Waiting   *LOCAL.INST45.050413230235

db2pd -db musicdb -transactions

Database Partition 0 -- Database MUSICDB -- Active -- Up 0 days 00:14:31

Transactions:
Address     AppHandl [nod-index] TranHdl    Locks     State   ...........  LogSpace
0x01DC1580  12       [000-00012] 3          2         WRITE   ...........  7322290
0x01DC2000  13       [000-00013] 5          0         READ    ...........  0
0x01DC2A80  21       [000-00021] 4          3         READ    ...........  0
```

图 3-18 定位事务句柄

```
db2pd -db musicdb -transactions

Database Partition 0 -- Database MUSICDB -- Active -- Up 0 days 00:14:31

Transactions:
Address     AppHandl [nod-index] TranHdl    Locks     State   ...........   LogSpace
0x01DC1580 12        [000-00012] 3          2         WRITE   ...........   7322290
0x01DC2000 13        [000-00013] 5          0         READ    ...........   0
0x01DC2A80 21        [000-00021] 4          3         READ    ...........   0
```

```
db2pd -db musicdb -locks

Database Partition 0 -- Database MUSICDB -- Active -- Up 0 days 05:14:31

Locks:
Address     TranHdl    Lockname                   Type       Mode Sts Owner Dur HldCnt Att Rlse
0x01DDE670 3          53514c4332453036bd4a32c841 Internal P ..S  G   3     1   0      0   0x0
0x01DDE738 4          53514c4332453036bd4a32c841 Internal P ..S  G   4     1   0      0   0x0
0x01DDE6C0 3          0d000400000000000000000054 Table      ..X  G   3     1   0      2   0x0
0x01DDE760 4          0d000400000000000000000054 Table      .IX  W   3     1   0      0   0x0
0x01DDE828 4          01000000010000000100f30056 Internal V ..S  G   4     1   0      0   0x0
0x01DDE7B0 4          53514c4445464c5428dd630641 Internal P ..S  G   4     1   0      0   0x0
```

图 3-19　根据事务句柄(TransHdl)来定位表上的加锁情况

例 3-3　诊断锁超时。

使用 db2pd -db sample -locks -transactions -applications -dynamic 命令可获取下列结果：

```
    锁定:
Address         TranHdl Lockname      Type Mode Sts Owner Dur HldCnt Att   ReleaseFlg
0x07800000202E5238 3       00020002000000040000000052 Row  ..X  G   3     1   0
0x0000  0x40000000
0x07800000202E4668 2       00020002000000040000000052 Row  ..X  W*  3     1   0
0x0000  0x40000000
```

对于使用-db 数据库名称选项指定的数据库，开头的结果会显示该数据库的锁定。您会发现 TranHdl 2 正在等待 TranHdl 3 挂起的锁定。其中 52 表示行锁，54 表示表锁，W* 表示锁等待超时，W 表示正在等待锁。

```
    事务:
Address            AppHandl [nod-index] TranHdl Locks State Tflag      Tflag2
Firstlsn         Lastlsn          LogSpace SpaceReserved TID            AxRegCnt GXID
0x0780000020251B80 11       [000-00011] 2       4     READ  0x00000000 0x00000000
0x000000000000 0x000000000000 0         0            0x0000000000B7 1        0
```

```
0x0780000020252900 12      [000-00012] 3      4     WRITE 0x00000000 0x00000000
0x000000FA000C 0x000000FA000C 113     154         0x0000000000B8 1        0
```

您会发现 TranHdl 2 与 AppHandl 11 相关联，而 TranHdl 3 与 AppHandl 12 相关联。

```
应用程序:
Address            AppHandl [nod-index] NumAgents CoorPid Status         C-AnchID
C-StmtUID L-AnchID L-StmtUID Appid
0x07800000006879E0 12      [000-00012] 1        1073336 UOW-Waiting   0        0
17       1         *LOCAL.burford.060303225602
0x0780000000685E80 11       [000-00011] 1        1040570 UOW-Executing 17      1
94       1         *LOCAL.burford.060303225601
```

您会发现 AppHandl 12 最后运行动态语句 17、1，ApplHandl 11 当前正在运行的动态语句是 17、1，而最后运行的语句是 94、1。

```
    动态 SQL 语句:
Address            AnchID StmtUID NumEnv NumVar NumRef NumExe Text
0x07800000209FD800 17     1       1      1      2      2     update SAMPLE set c1 = 5
0x07800000209FCCC0 94     1       1      1      2      2     set lock mode to wait 1
```

您会发现，文本(Text)列显示与锁定超时相关联的 SQL 语句。

例 3-4 使用-wlocks 选项捕获所有正在等待的锁定。

在下面的样本输出中，应用程序 1(AppHandl 47)正在执行插入操作，而应用程序 2 (AppHandl 46)正在选择该表：

```
prod@sapr3:/home/prod =>db2pd -db SAMPLE -locks wait
Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 04:20:59
Locks:
Address    TranHdl    Lockname                   Type       Mode Sts Owner      Dur
HoldCount  Att  ReleaseFlg
0x7F8A0DB0 7         020006000400040010000000052 Row        .NS  W   2          1
0          0x00 0x00000001  TbspaceID 2    TableID 6
   PartitionID 0 Page 320 Slot 4
0x7F8A1780 2         020006000400040010000000052 Row        ..X  G   2          1
0          0x00 0x40000000  TbspaceID 2    TableID 6
```

例 3-5 监控动态 SQL 语句。

db2pd -applications 命令报告动态 SQL 语句的当前及最后一个锚点标识和语句唯一标识，这允许直接从应用程序映射至动态 SQL 语句。

```
    db2pd -db sample -app -dyn
```

```
应用程序：
Address            AppHandl [nod-index] NumAgents  CoorPid Status
0x00000002006D2120 780      [000-00780] 1          10615   UOW-Executing—
应用程序状态
C-AnchID C-StmtUID  L-AnchID L-StmtUID  Appid
163      1          110      1          *LOCAL.burford.050202200412
动态 SQL 语句：
Address    AnchID StmtUID   NumEnv   NumVar   NumRef    NumExe  Text
0x0000000220A02760 163 1    2   2   2    1   select * from employee
0x0000000220A0B460 110  1   2    2  2    1   update employee set salary=1234
```

例 3-6 确定哪个应用程序使用了大量表空间。

通过使用 db2pd –tcbstats 命令，可以标识对表执行插入操作的次数。以下是用户定义的全局临时表 TEMP1 的样本信息：

```
TCB 表信息：
Address            TbspaceID TableID PartID MasterTbs MasterTab TableName
SchemaNm ObjClass DataSize  LfSize LobSize XMLSize
0x0780000020B62AB0 3         2       n/a    3         2         TEMP1     SESSION
Temp     966       0      0       0
TCB 表状态：
Address            TableName   Scans   UDI   PgReorgs   NoChgUpdts   Reads
FscrUpdates Inserts   Updates   Deletes   OvFlReads OvFlCrtes
   0x0780000020B62AB0 TEMP1       0       0     0          0           0       0
43968     0         0          0        0
```

然后，可以通过 db2pd -tablespaces 命令获取表空间 3 的信息。样本输出如下所示：

```
表空间 3 配置：
Address            Type Content PageSz ExtentSz Auto Prefetch BufID BufIDDisk FSC
NumCntrs MaxStripe  LastConsecPg Name
0x0780000020B1B5A0 DMS  UsrTmp  4096   32       Yes  32       1     1         On
1        0          31          TEMPSPACE2
表空间 3 统计信息：
Address            TotalPgs  UsablePgs  UsedPgs    PndFreePgs FreePgs    HWM
State     MinRecTime NQuiescers
0x0780000020B1B5A0 5000       4960       1088       0          3872       1088
0x00000000 0          0
表空间 3 自动调整大小统计信息：
Address            AS  AR  InitSize   IncSize   IIP MaxSize   LastResize   LRF
0x0780000020B1B5A0 No  No  0          0         No  0         None         No
容器：
Address            ContainNum Type    TotalPgs   UseablePgs StripeSet  Container
```

```
0x0780000020B1DCC0 0     File    5000    4960    0 /home/db2inst1/tempspace2a
```

您会注意到通过引用 FreePgs 列填满的空间。因为可用页数下降，所以可用空间减少。还应注意，FreePgs 加上 UsedPgs 的值将等于 UsablePgs 的值。

一旦了解了这一点，您就可以标识使用表 TEMP1 的动态 SQL 语句：

```
db2pd -db sample -dyn
数据库分区 0 -- 数据库 SAMPLE -- 活动 -- 正常运行 0 天 00:13:06
动态高速缓存:
当前使用的内存              1022197
堆总大小                     1271398
高速缓存溢出标志             0
引用数                       237
语句插入数目                 32
语句删除数目                 13
变动插入数目                 21
语句数目                     19
动态 SQL 语句:
Address            AnchID StmtUID  NumEnv  NumVar  NumRef  NumExe  Text
0x0000000220A08C40 78     1        2       2       3       2       declare global
temporary table temp1 (c1 char(6)) not logged
0x0000000220A8D960 253    1        1       1       24      24      insert into
session.temp1 values('TEST')
```

最后，可以将它映射至 db2pd -app 输出以标识应用程序：

```
    应用程序:
    Address            AppHandl [nod-index] NumAgents  CoorPid Status
    0x0000000200661840 501      [000-00501] 1          11246   UOW-Waiting
    C-AnchID C-StmtUID  L-AnchID L-StmtUID  Appid
       0        0        253      1          *LOCAL.db2inst1.050202160426
```

针对先前使用的 db2pd 中的动态 SQL 语句的请求产生的锚点标识(AnchID)，将与针对关联应用程序的请求一起使用。应用程序结果显示最后一个锚点标识(L-AnchID)与该锚点标识(AnchID)相同。一次运行 db2pd 产生的结果将在下一次运行 db2pd 时使用。

db2pd -agent 的输出将显示应用程序读取的行数(Rowsread 列)和写入的行数(Rowswrtn 列)。这些值将显示应用程序已完成的部分及尚未完成的部分：

```
    Address            AppHandl [nod-index] AgentPid  Priority  Type DBName
    0x0000000200698080 501      [000-00501] 11246     0         Coord SAMPLE
    State        ClientPid  Userid   ClientNm Rowsread  Rowswrtn  LkTmOt
    Inst-Active 26377      db2inst1 db2bp    22        9588      NotSet
```

db2pd -agent 请求中的 AppHandl 和 AgentPid 的值可映射回 db2pd -app 请求中的 AppHandl 和 CoorPid 的对应值。

如果您怀疑内部临时表占满了表空间，那么上面这些步骤会稍有不同。仍然可以使用 db2pd -tcbstats 来标识具有最大插入数目的表。以下是隐式临时表的样本信息：

```
TCB 表信息：
Address           TbspaceID TableID PartID MasterTbs MasterTab TableName
SchemaNm ObjClass        DataSize  ...
0x0780000020CC0D30 1        2      n/a    1         2        TEMP (00001,00002)
<30>     <JMC Temp       2470      ...
0x0780000020CC14B0 1        3      n/a    1         3        TEMP (00001,00003)
<31>     <JMC Temp       2367      ...
0x0780000020CC21B0 1        4      n/a    1         4        TEMP (00001,00004)
<30>     <JMC Temp       1872      ...
TCB 表状态：
Address           TableName         Scans   UDI   PgReorgs   NoChgUpdts Reads
FscrUpdates Inserts ...
0x0780000020CC0D30 TEMP (00001,00002) 0     0   0       0       0     0 43219 ...
0x0780000020CC14B0 TEMP (00001,00003) 0     0   0       0       0     0 42485 ...
0x0780000020CC21B0 TEMP (00001,00004) 0     0   0       0       0     0
          ......................略..........................
```

在此例中，使用命名约定“TEMP (TbspaceID, TableID)”的表中有大量插入。这些是隐式临时表。SchemaNm 列中的值的命名约定为 AppHandl 的值与 SchemaNm 的值并置，这使得它能够标识正在工作的应用程序。

然后，可以将这些信息映射至 db2pd -tablespaces 产生的输出，以查看表空间 1 的已使用空间。在表空间统计信息中记下已使用的页数与可用页数之间的关系。

```
    表空间配置：
    Address              Id    Type Content PageSz ExtentSz Auto Prefetch BufID
BufIDDisk FSC NumCntrs MaxStripe LastConsecPg Name
    0x07800000203FB5A0 1     SMS  SysTmp  4096   32       Yes  320      1     1
On  10        0         31          TEMPSPACE1
    表空间统计信息：
    Address              Id    TotalPgs   UsablePgs  UsedPgs    PndFreePgs FreePgs
HWM    State      MinRecTime NQuiescers
    0x07800000203FB5A0 1     6516       6516       6516       0          0
0     0x00000000 0          0
    表空间自动调整大小统计信息：
    Address         Id     AS  AR  InitSize    IncSize    IIP MaxSize    LastResize LRF
    0x07800000203FB5A0 1     No  No  0         0          No  0          None         No
```

```
    容器:
        ...
```

然后，可以使用命令 db2pd -app 标识应用程序句柄 30 和 31(因为它们出现在-tcbstats 输出中)：

```
应用程序:
Address            AppHandl [nod-index] NumAgents  CoorPid    Status
C-AnchID C-StmtUID  L-AnchID   L-StmtUID  Appid
0x07800000006FB880 31       [000-00031] 1          4784182    UOW-Waiting     0
0          107        1          *LOCAL.db2inst1.051215214142
0x07800000006F9CE0 30       [000-00030] 1          8966270    UOW-Executing  107
1          107        1          *LOCAL.db2inst1.051215214013
```

最后，使用 db2pd -dyn 命令将它映射至动态 SQL：

```
    动态 SQL 语句:
Address            AnchID StmtUID    NumEnv     NumVar     NumRef     NumExe Text
0x0780000020B296C0 107    1          1          1          43         43
select c1, c2 from test group by c1,c2
```

例 3-7　监视全表扫描的表。

发出 db2pd -tcbstats 命令，可以查找发生过的全表扫描的表，如下所示：

```
    C:\> db2 connect to sample
    数据库连接信息
    数据库服务器 = DB2/AIX64 11.1.1.1
     SQL 授权标识 = DB2INST1
    本地数据库别名 = SAMPLE
     C:\> db2 select * from test1 -------test1 上没有索引，执行一次全表扫描
     ID
     - - - - - - - - - - - - - - - - - - - - -
     1
     1
    2 条记录已选择。
     C:\> db2 select * from test1 -------test1 上没有索引，执行第二次全表扫描
     ID
     - - - - - - - - - - - - - - - - - - - - -
     1
     1
    2 条记录已选择。
     C:\> db2 select * from test2 -------test2 上没有索引，执行一次全表扫描
     ID NAMEJL
     - - - - - - - - - - - - - - - - - - - - -
```

```
1 rhetteaaa
2 danielbbb
2 条记录已选择。
C:\> db2pd -db sample -tcbstats
Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:15:24
TCB Table Information:
Address TbspaceID TableID PartID MasterTbs MasterTab TableNameSchemaNm
..........................................略......................
0x7EAF1920 0 7 n/a 0 7 SYSINDEXES SYSIBM
0x7EAF38A0 3 6 n/a 3 6 TEST1 DB2INST1 ------表 TEST1 存在全表扫描
0x7EAF41A0 3 7 n/a 3 7 TEST2 DB2INST1 ------表 TEST2 存在全表扫描
.........................略...........................................
0x7EAABBA0 0 13 n/a 0 13 SYSPLAN SYSIBM
...........................略.......................
TCB Table Stats: -----查看全表扫描的详细信息
Address TableName Scans UDI RTSUDI PgReorgs
.........................略.................
0x7EAF1920 SYSINDEXES 0 309 309 0
0x7EAF38A0 TEST1 2 0 0 0 ---表 TEST1 全表扫描两次
0x7EAF41A0 TEST2 1 0 0 0 ---表 TEST2 全表扫描一次
0x7EAABBA0 SYSPLAN 0 2 2 0
..............................略.............................
```

命令成功完成，部分输出结果如上所示。查看都有哪些表发生了表扫描，可以看输出结果中的“TCB Table Information”部分，TableName 部分显示了发生过全表扫描的表的名字。如果要查看全表扫描发生的次数，可以查看 TCB Table Stats 部分。

例 3-8　监视恢复。

命令 db2pd -recovery 显示了几个可用于验证是否正在进行恢复的计数器：当前日志和当前 LSN 提供了日志位置，已完成的工作，计算目前已完成的字节数。

```
恢复:
恢复状态          0x00000401
当前日志           S0000005.LOG
当前 LSN           000002551BEA
作业类型           ROLLFORWARD RECOVERY
作业标识           7
作业开始时间      (1107380474) Wed Feb  2 16:41:14 2014
作业描述          数据库前滚恢复
调用程序类型      用户
总阶段            2
当前阶段          1
进度:
```

```
地址                阶段号 描述  开始时间                   已完成的工作  总工作
0x0000000200667160 1    向前  Wed Feb  2 16:41:14 2005  2268098 字节 未知
0x0000000200667258 2    向后  NotStarted                0 字节       未知
```

例 3-9 确定事务正在使用的资源量。

命令 db2pd -transactions 提供了锁定数、第一个日志序号(LSN)、最后一个 LSN、已使用的日志空间和保留空间。这对于了解事务行为很有用。

```
事务：
Address            AppHandl [nod-index] TranHdl  Locks  State  Tflag
0x000000022026D980 797      [000-00797] 2        108    WRITE  0x00000000
0x000000022026E600 806      [000-00806] 3        157    WRITE  0x00000000
0x000000022026F280 807      [000-00807] 4        90     WRITE  0x00000000

Tflag2     Firstlsn       Lastlsn        LogSpace   SpaceReserved
0x00000000 0x000001072262 0x0000010B2C8C 4518       95450
0x00000000 0x000001057574 0x0000010B3340 6576       139670
0x00000000 0x00000107CF0C 0x0000010B2FDE 3762       79266

TID            AxRegCnt   GXID
0x000000000451  1          0
0x0000000003E0  1          0
0x000000000472  1          0
```

例 3-10 监视日志使用情况。

命令 db2pd -logs 对于监视数据库的日志使用情况很有用。通过观察已写入的页面输出，可以确定是否正在使用日志。

```
Logs:
Current Log Number            11
Pages Written                5698
Cur Commit Disk Log Reads     0
Cur Commit Total Log Reads   154
Method 1 Archive Status       n/a
Method 1 Next Log to Archive  n/a
Method 1 First Failure        n/a
Method 2 Archive Status       n/a
Method 2 Next Log to Archive  n/a
Method 2 First Failure        n/a
Log Chain ID                 0
Current LSN                  0x0000006721A46951

Address            StartLSN       State      Size       Pages      Filename
```

```
0x07000000A004C290 0000006714410010 0x00000000 16380  16380   S0000008.LOG
0x07000000A004CAF0 000000671840C010 0x00000000 16380  16380   S0000009.LOG
0x07000000A004D350 000000671C408010 0x00000000 16380  16380   S0000010.LOG
0x07000000A004DBB0 0000006720404010 0x00000000 16380  16380   S0000011.LOG
0x07000000A004E410 0000006724400010 0x00000000 16380  16380   S0000012.LOG
```

使用此输出可以确定两个问题：

- 如果归档存在问题，那么 Archive Status 将显示为 Failure，表示最近的日志归档失败。如果正在发生的归档失败导致根本无法归档日志，那么将显示 First Failure。
- 如果日志归档速度非常慢，那么下一个要归档的日志值将小于当前日志编号。这可能导致填满日志路径，从而在完全填满日志路径时防止数据库中出现任何数据更改。

例 3-11　收集操作系统信息。

通过发出 db2pd-osinfo 命令，我们可以方便地获取操作系统的一些配置信息和运行信息：

```
/home/db2inst1/sqllib/db2dump$db2pd -osinfo
Operating System Information:
OSName:   AIX
NodeName: RBPPTRN
Version:  6
Release:  1
Machine:  000D655AD400

CPU Information:
TotalCPU    OnlineCPU   ConfigCPU   Speed(MHz)  HMTDegree  Cores/Socket
64          16          64          3000        4          n/a

Physical Memory and Swap (Megabytes):
TotalMem    FreeMem     AvailMem    TotalSwap   FreeSwap
32768       3873        n/a         16384       12270

Virtual Memory (Megabytes):
Total       Reserved    Available   Free
49152       n/a         n/a         16143

Message Queue Information:
MsgSeg      MsgMax      MsgMap      MsgMni      MsgTql     MsgMnb      MsgSsz
n/a         4194304     n/a         n/a         n/a        4194304     n/a

Shared Memory Information:
ShmMax                ShmMin                ShmIds      ShmSeg
68719476736            1                    131072      0
```

```
    Semaphore Information:
    SemMap      SemMni     SemMns      SemMnu      SemMsl      SemOpm      SemUme
SemUsz      SemVmx      SemAem
    n/a        131072     n/a        n/a        65535       1024        n/a
n/a         32767      16384

    CPU Load Information:
    Short     Medium    Long
    2.045822  1.411087  1.220459

    CPU Usage Information (percent):
    Total       Usr         Sys         Wait      Idle
    12.625000 11.312500 0.500000  0.812500  87.375000
```

从上面收集的信息我们可以看到操作系统的版本、CPU 配置、物理内存、虚拟内存、消息队列、共享内存、信号灯等信息。其中上面列出的消息队列、共享内存、信号灯都是与该 DB2 实例相关的信息。

例 3-12　使用 db2pd -dbmcfg 命令，收集数据库管理器配置参数。

```
    /home/db2inst1$ db2pd -dbmcfg
    Database Partition 0 -- Active -- Up 0 days 01:29:15
    Database Manager Configuration Settings:
    Description                Memory Value             Disk Value
    RELEASE                     0xa00                    0xa00
    CPUSPEED                    5.825579e-07             5.825579e-07
    COMM BANDWIDTH             1.000000e+02             1.000000e+02
    NUMDB                      8                        8
    DATALINKS                  NO                       NO
    FEDERATED                  YES                      YES
    ......
```

从中我们可以看到数据库管理器的运行时间、所有的配置参数，其中 Memory Value 为当前值，Disk Value 为下一次启动 DB2 实例生效的值。

例 3-13　收集数据库配置参数。

使用-dbcfg 可以获得数据库配置参数，配合-db <dbname>选项使用可以获取不同数据库的配置参数。注意，使用该命令收集数据库配置参数时需要该数据库处于活动状态。例如：

```
    /home/db2inst1$ db2pd -db sample -dbcfg
    Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:00:03
    Database Configuration Settings:
```

```
Description Memory Value Disk Value
DB configuration release level 0xa00 0xa00
Database release level 0xa00 0xa00
Database territory US US
Database code page 819 819
Database code set ISO8859-1 ISO8859-1
Database country/region code 1 1
Database collating sequence UNIQUE UNIQUE
ALT_COLLATE NON_UNIQUE NON_UNIQUE
DYN QUERY MGMT DISABLE DISABLE
......
```

同样，Memory Value 为当前值，Disk Value 为下一次启动该数据库时生效的值。

例 3-14　监控缓冲池信息。

缓冲池设置对于数据库的性能影响非常大。可以使用-bufferpools 选项来获取数据库的缓冲池信息。该选项同样需要配合-db <dbname>配合一起使用，例如：

```
/home/db2inst1$ db2pd -db sample -bufferpools
Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:04:53
BufferPools:
First Active Pool ID 1
Max Bufferpool ID 1
Max Bufferpool ID on Disk 1
Num Bufferpools 5
Address Id Name PageSz PA-NumPgs BA-NumPgs
BlkSize ES NumTbsp PgsLeft CurrentSz PostAlter SuspndTSCt
0x0780000020332F20 1    IBMDEFAULTBP 4096 1000 0  0 N 6 0 1000 1000 0
0x07800000203320A0 4096 IBMSYSTEMBP4K 4096 16 0  0 N 0 0 16 16 0
0x0780000020332440 4097 IBMSYSTEMBP8K 8192 16 0  0 N 0 0 16 16 0
0x07800000203327E0 4098 IBMSYSTEMBP16K 16384 16 0  0 N 0 0 16 16 0
0x0780000020332B80 4099 IBMSYSTEMBP32K 32768 16 0  0 N 0 0 16 16 0
```

注意：

其中，IBMSYSTEMBP4K、IBMSYSTEMBP8K、IBMSYSTEMBP16K、IBMSYSTEMBP32K 为 DB2 默认创建的系统缓冲池，在之前这四个缓冲池被称为“隐藏缓冲池”。如果在 db2diag.log 中看到使用这些缓冲池的消息，请检查缓冲池的设置是否合理，是否设置得超过了操作系统的内存上限。因为这些缓冲池都非常小，如果使用到这些缓冲池的话，性能一定会非常差。

例 3-15 监控表空间信息。

可以使用-tablespaces 选项获得表空间信息，同样需要配合-db <dbname>选项使用：

```
$ db2pd -db sample -tablespaces
Tablespace Autoresize Statistics:
Address     Id AS  AR  InitSize IncSize IIP MaxSize LastResize  LRF
0x05267580 0 Yes Yes  33554432 -1      No  None    None        No
0x05267D50 1 Yes No   0         0      No  0       None        No
0x0526C510 2 Yes Yes  33554432 -1      No  None    None        No
0x0526CCE0 3 Yes Yes  33554432 -1      No  None    None        No
0x05C73EE0 4 Yes Yes  33554432 -1      No  None    None        No
0x05C98360 5 Yes No   0         0      No  0       None        No
```

在命令输出的 Tablespace Autoresize Statistics 段中，AS 字段表明表空间是否启用了自动存储器，Yes 为启用，No 为未启用。AR 表示 Automatic Resize。

例 3-16 监控 reorg 的进度。

使用–reorg 选项可以监控表 reorg 的进度，如下所示：

```
db2 reorg table sales1
db2pd -db sample -reorg
Database Partition 0 -- Database SAMPLE -- Active -- Up 0 days 00:20:06
Table Reorg Stats:
Address TbspaceID TableID TableName Start End
0x07800000294CD438 3 259 SALES1 2013-12-19 15:06:05.271603 n/a
PhaseStart MaxPhase Phase CurCount MaxCount Type Status Completion IndexID
TempSpaceID
2013-12-19 15:06:05.565690 2 Build 470 470 Offline Started 0 0 3
```

3.5.2 使用 db2pd 监控死锁案例

死锁经常会存在于我们的应用系统中，如何捕获死锁信息并解决死锁问题，是比较复杂的问题。除了抓取事件监视器，DB2 也可以使用 db2pd 命令和 db2cos 脚本来获取死锁信息，提供了一种新的途径来诊断死锁问题。

我们可以使用 db2pdcfg -catch 命令来捕获错误信息，然后调用 sqllib/db2cos 脚本来收集出错时的现场信息。该命令的使用语法如下：

```
Usage:
   -catch clear | status | <errorCode> [<action>] [count=<count>]
      Sets catchflag to catch error or warning.

      Error Codes:
      <sqlCode>[,<reasonCode>] / sqlcode=<sqlCode>[,<reasonCode>]
```

```
        ZRC (hex or integer)
        ZRC #define (such as SQLP LTIMEOUT)
        ADM (such as ADM1611)
        String (such as diagstr="Hello World")
        ECF (hex or integer)
        "deadlock" or "locktimeout"

        Actions:
        [stack] (default)            Produce stack trace in db2diag.log
        [db2cos] (default)           Run sqllib/db2cos callout script
        [stopdb2trc]                 Stop db2trc
        [dumpcomponent]              Dump component flag
        [component=<componentID>]    Component ID
        [lockname=<lockname>]        Lockname for catching specific lock
            (lockname=000200030000001F0000000052)
        [locktype=<locktype>]        Locktype for catching specific lock
            (locktype=R or locktype=52)
```

下面我们通过实例来讲解如何使用 db2pdcfg -catch 命令获取死锁信息。如无特殊说明，命令均使用 DB2 实例用户执行。

(1) 将$HOME/sqllib/cfg/db2cos 示例脚本复制到$HOME/sqllib 下，并改变属性，为实例用户添加执行权限：

```
cp $HOME/sqllib/cfg/db2cos $HOME/sqllib
chmod u+x $HOME/sqllib/db2cos
```

(2) 使用 db2pdcfg -catch 捕获死锁信息，当死锁出现的时候调用 db2cos 命令。可以使用如下命令之一：

```
1)db2pdcfg -catch deadlock
2)db2pdcfg -catch -911,2
```

输出如下：

```
Error Catch #2
   Sqlcode:         -911
   ReasonCode:       2
   ZRC:             0
   ECF:             0
   Component ID:     0
   LockName:        Not Set
   LockType:        Not Set
   Current Count:    0
```

```
   Max Count:     255
   Bitmap:        0x661
   Action:        Error code catch flag enabled
   Action:        Execute sqllib/db2cos callout script
   Action:        Produce stack trace in db2diag.log
```

此时查看 db2diag.log 的输出，可以看到类似信息：

```
2016-12-24-14.46.20.359000+480 I474200H346       LEVEL: Event
PID     : 3768                 TID  : 4480       PROC : db2syscs.exe
INSTANCE: DB2 01                NODE : 000
EDUID   : 4480                 EDUNAME: db2pdbe 0
FUNCTION: DB2 UDB, RAS/PD component, pdErrorCatch, probe:30
START   : Error catch set for sqlCode -911 reasonCode 2
```

我们可以看到错误捕获机制已经启动。

(3) 新打开命令窗口，我们称之为窗口 1，输入如下命令：

```
$db2 +c
db2 => connect to sample
Database Connection Information
Database server = DB2/AIX64 11.1.1.1
SQL authorization ID = DB2INST1
Local database alias = SAMPLE
db2 => create table tstdlock1 (id int, name char(10))
DB20000I The SQL command completed successfully.
db2 => commit
DB20000I The SQL command completed successfully.
db2 => insert into tstdlock1 values(1,'test1')
DB20000I The SQL command completed successfully.
```

(4) 再新开一个命令窗口，我们称之为窗口 2，输入如下命令：

```
$db2 +c
db2 => connect to sample
Database Connection Information
Database server = DB2/AIX64 11.1.1.1
SQL authorization ID = DB2INST1
Local database alias = SAMPLE
db2 => create table tstdlock2 (id int, name char(10))
DB20000I The SQL command completed successfully.
db2 => commit
DB20000I The SQL command completed successfully.
db2 => insert into tstdlock2 values(2,'test2')
DB20000I The SQL command completed successfully.
```

```
db2 => select * from tstdlock1
```

此时该命令会挂起，处于锁等待状态，等待窗口 1 中的 insert 语句提交后才能继续进行。

(5) 切换到窗口 1，输入如下命令：

```
db2 => select * from tstdlock2
```

此时该命令也会挂起，处于锁等待状态，等待窗口 2 中的 insert 语句提交后才能继续进行。

(6) 等待一段时间，取决于数据库配置参数 DLCHKTIME 的设置，默认为 10 秒就会发现窗口 2 中的事务因为死锁而回滚：

```
SQL0911N The current transaction has been rolled back because of a deadlock
or timeout. Reason code "2". SQLSTATE=40001
```

而窗口 1 中的命令执行成功：

```
db2 => select * from tstdlock2
ID             NAME
----------- ----------
0 record(s) selected.
```

注意：

在实际的测试中，也可能是窗口 1 中的事务回滚。

此时查看 db2diag.log 文件，会看到如下信息：

```
2016-12-24-14.47.20.359000+480 I28093C411 LEVEL: Event
PID: 1597606 TID : 1 PROC : db2agent (SAMPLE) 0
INSTANCE: db2inst1 NODE : 000 DB : SAMPLE
APPHDL: 0-8 APPID: *LOCAL.db2inst1.060330092553
FUNCTION: DB2 UDB, trace services, pdInvokeCalloutScript, probe:10
START: Invoking sqllib/db2cos script from global services sqlzeMapZrc
2016-12-24-14.47.21.359000+480 I28505C388 LEVEL: Event
PID: 1597606 TID : 1 PROC : db2agent (SAMPLE) 0
INSTANCE: db2inst1 NODE : 000 DB : SAMPLE
APPHDL: 0-8 APPID: *LOCAL.db2inst1.060330092553
FUNCTION: DB2 UDB, trace services, pdInvokeCalloutScript, probe:20
STOP: Completed invoking sqllib/db2cos script
```

这说明我们的错误捕获机制已经成功捕获到死锁信息，并且调用了 db2cos 脚本。此时查看$HOME/sqllib/db2dump 目录，会看到 db2cos.rpt 文件。注意，对于出现死锁的情况，

我们的错误捕获机制会两次调用 db2cos 脚本，第一次是在事务回滚前，第二次则是在事务回滚后。查看死锁的信息，我们应该关注第一次调用 db2cos 脚本的输出。

现在我们看一下 db2cos 例子脚本的内容，可以看到出现死锁时我们的处理机制：

```
echo "Lock Deadlock Caught" >> $HOME/sqllib/db2dump/db2cos.rpt
date >> $HOME/sqllib/db2dump/db2cos.rpt
echo "Instance " $instance >> $HOME/sqllib/db2dump/db2cos.rpt
echo "Datbase: " $database >> $HOME/sqllib/db2dump/db2cos.rpt
echo "Partition Number: " $dbpart >> $HOME/sqllib/db2dump/db2cos.rpt
echo "PID: " $pid >> $HOME/sqllib/db2dump/db2cos.rpt
echo "TID: " $tid >> $HOME/sqllib/db2dump/db2cos.rpt
echo "Function: " $function >> $HOME/sqllib/db2dump/db2cos.rpt
echo "Component: " $component >> $HOME/sqllib/db2dump/db2cos.rpt
echo "Probe: " $probe >> $HOME/sqllib/db2dump/db2cos.rpt
echo "Timestamp: " $timestamp >> $HOME/sqllib/db2dump/db2cos.rpt
echo "AppID: " $appid >> $HOME/sqllib/db2dump/db2cos.rpt
echo "AppHdl: " $apphld >> $HOME/sqllib/db2dump/db2cos.rpt
db2pd -db $database >> $HOME/sqllib/db2dump/db2cos.rpt
```

我们看到，如果捕获到的错误是死锁，我们将执行 db2pd -db $database 命令来获取数据库的所有 db2pd 输出。您可以更改该脚本，从而仅获取部分输出，或者获取其他信息。

我们查看一下 db2cos.rpt 文件，可以看到数据库中存在的锁信息：

```
Locks:
Address TranHdl Lockname Type Mode Sts Owner Dur HldCnt Att Rlse
0x402C0740 2 53514C4332453036C8324ABC41 Internal P ..S G 2 1 0 0 0x40
0x402C0D08 3 53514C4332453036C8324ABC41 Internal P ..S G 3 1 0 0 0x40
0x402C06F0 2 00020012000000040000000052 Row ..X G 2 1 0 8 0x40
0x402C0808 3 00020012000000040000000052 Row .NS W* 3 1 0 0 0x0
0x402C09C0 2 0000000100000000100012E0056 Internal V ..S G 2 1 0 0 0x40
0x402C0678 3 000000010000000010001C40056 Internal V ..S G 3 1 0 0 0x40
0x402C1000 3 00020013000000040000000052 Row ..X G 3 1 0 8 0x40
0x402C0AB0 2 00020013000000040000000052 Row .NS W 3 1 0 0 0x0
0x402C0FB0 3 00020013000000000000000054 Table .IX G 3 1 0 0 0x40
0x402C0F38 2 00020013000000000000000054 Table .IS G 2 1 0 0 0x0
0x402C0B78 2 00020012000000000000000054 Table .IX G 2 1 0 0 0x40
0x402C1028 3 00020012000000000000000054 Table .IS G 3 1 0 0 0x0
```

注意状态为 W*的锁，这个锁就是最后回滚的那个应用程序的锁。上面的锁信息为：

```
0x402C0808 3 00020012000000040000000052 Row .NS W* 3 1 0 0 0x0
```

我们看到事务句柄 TranHdl 为 3，同时我们看到持有这个锁 00020012000000040000000052

的另一个应用程序的句柄为 2：

```
    0x402C06F0 2 00020012000000040000000052 Row ..X G 2 1 0 8 0x40
```

使用 db2pd –db sample transactions 查看事务信息，我们可以看到应用程序句柄 7 对应事务句柄 2，应用程序句柄 8 对应事务句柄 3：

```
    Transactions:
    Address AppHandl [nod-index] TranHdl Locks State Tflag Tflag2 Firstlsn
Lastlsn LogSpace SpaceReserved TID AxRegCnt GXID
    0x4024D580 7 [000-00007] 2 6 WRITE 0x00000000 0x00000 000 0x0000059D89F4
0x0000059D89F4 108 170 0x000000000B08 1 0
    0x4024E000 8 [000-00008] 3 6 WRITE 0x00000000 0x00000 000 0x0000059DA3E0
0x0000059DA3E0 108 170 0x000000000B22 1 0
```

此时我们使用 db2pd –db sample -appl 查看应用程序的信息，可以获得 C-AnchID 和 C-StmtUID：

```
    Applications:
    Address AppHandl [nod-index] NumAgents CoorPid Status C-AnchID C-StmtUID
L-AnchID L-StmtUID Appid
    0x300E3DD0 8 [000-00008] 1 1597606 UOW-Executing 196 1 131 1
*LOCAL.db2inst1.060330092553
    0x300E2B60 7 [000-00007] 1 1990788 Lock-wait 46 1 46 1
*LOCAL.db2inst1.060330092132
```

同时我们使用 db2pd –db sample -dynamic 还可以查看动态语句的输出，可以根据 C-AnchID 和 C-StmtUID 找到当前正在执行的 SQL 语句：

```
    Dynamic SQL Statements:
    Address AnchID StmtUID NumEnv NumVar NumRef NumExe Text
    0x409D80C0 9 1 2 3 4 2 create table tstdlock2 (id int, name char(10))
    0x409E0A70 46 1 1 1 2 2 select * from tstdlock2
    0x409DED70 97 1 1 1 1 1 SELECT COUNT(*) FROM SYSCAT.PROCEDURES WHERE PROCNAME=
    'SYSINSTALLROUTINES'
    0x409D8BB0 131 1 1 2 3 3 insert into tstdlock2 values(2, 'test2')
    0x409D3A70 132 1 2 2 2 1 create table tstdlock1 (id int, name char(10))
    0x409D3F60 150 1 1 1 1 1 insert into tstdlock1 values(1, 'test1')
    0x409DFEF0 196 1 1 1 3 3 select * from tstdlock1
    0x409D3720 243 1 0 0 0 0 SET CURRENT LOCALE LC_CTYPE = 'zh_CN'
```

根据 AnchID 和 StmtUID，我们可以看到执行的语句为：

```
    0x409E0A70 46 1 1 1 2 2 select * from tstdlock2
```

```
0x409DFEF0 196 1 1 1 3 3 select * from tstdlock1
```

一旦我们定位引发死锁的语句，就可以根据业务逻辑对该 SQL 语句进行调优和创建合理的索引。

3.5.3 db2pd 使用问题总结

db2pd 是一款功能非常强大的工具，我们在使用过程中常见的有以下几个问题：

1) 因为 db2pd 一直在增强，所以在多个 DB2 版本之间在一些命令选项上有微小的差异。常常是一些命令选项在新的版本中已经变化或者添加了新的选项，希望读者使用时多使用 db2pd –h 查看帮助。

2) 因为 db2pd 工具可从 DB2 内存集迅速返回即时信息，所以该工具可用于故障诊断。该工具不需要获得任何锁存器或使用任何引擎资源就可以收集信息。因此，在 db2pd 收集信息时，有可能(并且预计)会检索到正在更改的信息；这样的话，数据可能不是十分准确。如果遇到正在更改的内存指针，可使用信号处理程序来防止 db2pd 异常终止。这可能会导致输出中出现诸如以下的消息："正在更改的数据结构已强制终止命令"。虽然如此，该工具对于故障诊断却非常有用。在不加锁的情况下收集信息有两个好处：检索速度更快并且不会争用引擎资源。

3) db2pd 除了具有监控功能之外，还可以用于故障诊断。如果要在出现特定 SQLCODE、ZRC 代码或 ECF 代码时捕获关于数据库管理系统的信息，那么可以使用 db2pdcfg -catch 命令完成此操作。捕获到错误时，将启动 db2cos(调出脚本)。db2cos 文件可以自行改变，以便运行解决问题所需的任何 db2pd 命令、操作系统命令或任何其他命令。在 UNIX 和 Linux 上，模板文件 db2cos 位于 sqllib/bin 中。在 Windows 操作系统上，db2cos 位于$DB2PATH\bin 目录中。

4) db2pd 命令的输出通常比较多，建议读者把输出结果管道输出到文件中，使用编辑器来查看。

5) 对于经常使用的 db2pd 选项，建议大家设置 db2pdopt 命令选项，这样可以在使用时快速调用 db2pdopt 设置的选项。下面我们举一个实际的例子：假如我们经常使用 db2pd -db sample -locks -transactions -applications –dynamic 来监控相关信息，可以像下面这样来设置。

在 UNIX/Linux 环境中，如果 shell 是 ksh，执行：

```
(1) export DB2PDOPT="-db sample -locks -transactions -applications -dynamic"
(2) db2pd
```

在 Windows 环境中，执行：

```
(1) set DB2PDOPT=-db sample -locks -transactions -applications -dynamic
(2) db2pd
```

执行上面的第(2)步之后，db2pd 命令的输出结果与发出了 db2pd -db sample -locks -transactions -applications –dynamic 命令的相同，也就是说 DB2PDOPT 定义了 db2pd 命令后使用的参数选项。这样便于快速调用 db2pd 来监控我们需要的信息。

3.6　内存监控

3.6.1　db2pd 内存监控

我们在查看数据库分区的内存统计信息时，可以通过使用 db2pd 的选项进行查看。

-dbptnmem

db2pd -dbptnmem 命令显示 DB2 服务器当前消耗的内存量，并在较高级别显示使用这些内存的服务器区域。

以下是在 AIX 机器上运行 db2pd -dbptnmem 的输出示例：

```
Database Partition Memory Controller Statistics

Controller Automatic: Y
Controller License Limit: N
Controller Limit Enforced: N
Memory Limit:          29564116KB
Current usage:          5408640KB
HWM usage:             7934080KB
Cached memory:          1169216KB
```

关于这些数据字段和列的描述如下：

- 控制器自动：指示内存控制器设置。如果 instance_memory 配置参数设置为 automatic，那么它将显示值“Y”。这意味着数据库管理器自动确定内存耗用量的上限。
- 内存限制：如果强制施加了实例内存限制，那么 instance_memory 配置参数的值是可以耗用的 DB2 服务器内存的上限。
- 当前使用量：服务器当前耗用的内存量。
- HWM 使用量：自激活数据库分区(在 db2start 命令运行时)以来消耗的内存高水位标记(HWM)或峰值。

- 高速缓存的内存：当前使用量中未使用但为了提高将来内存请求的性能而高速缓存的内存量。

```
Individual Memory Consumers:

Name            Mem Used (KB) HWM Used (KB) Cached (KB)
========================================================
APPL-PMBSDB         160000        160000      153280
DBMS-db2pmbs         55424         55424       15168
FMP RESOURCES        22528         22528           0
PRIVATE              80128        106944       17472
DB-PMBSDB          5090560       7617024      983296
```

上面列示了 DB2 服务器中所有已注册的内存“使用者”以及它们消耗的内存总量。列描述如下：

- 名称：内存“使用者”的简短专有名称，例如 APPL-dbname 为数据库 dbname 耗用的应用程序内存。
- DBMS-name：全局数据库管理器内存需求。
- FMP_RESOURCES：与 db2fmps 进行通信所需的内存。
- PRIVATE：其他专用内存需求。
- FCM_RESOURCES：快速通信管理器资源。
- LCL-pid：用于与本地应用程序进行通信的内存段。
- DB-dbname：为数据库 dbname 耗用的数据库内存。
- 使用的内存(KB)：当前分配给使用者的内存量。
- 使用的 HWM(KB)：使用者曾耗用的内存量的高水位标记(HWM)，即内存量峰值。
- 已高速缓存(KB)：在“使用的内存”中，当前未使用但立即可用于将来的内存分配的内存量。

-memblock

在尝试了解内存使用情况时，db2pd -memblock 命令非常有用，如下所示：

```
DBMS 集中的所有内存块：
地址                 池标识   池名称    块存活时间  大小(字节)  I  LOC   文件
0x0780000000740068  62     resynch     2        112        1  1746  1583816485
0x0780000000725688  62     resynch     1        108864     1  127   1599127346
0x07800000001F4348  57     ostrack     6        5160048    1  3047  698130716
0x07800000001B5608  57     ostrack     5        240048     1  3034  698130716
0x07800000001A0068  57     ostrack     1        80         1  2970  698130716
0x07800000001A00E8  57     ostrack     2        240        1  2983  698130716
```

```
0x07800000001A0208 57     ostrack     3       80        1   2999   698130716
0x07800000001A0288 57     ostrack     4       80        1   3009   698130716
0x0780000000700068 70     apmh        1      360        1   1024   3878879032
0x07800000007001E8 70     apmh        2      48         1   914    1937674139
0x0780000000700248 70     apmh        3      32         1   1000   1937674139
...
接下来是已排序的“性能池”输出
按 ostrack 池大小排序的内存块:
池标识     池名称   总大小(字节)        总计数      LOC    文件
57         ostrack    5160048           1         3047   698130716
57         ostrack    240048            1         3034   698130716
57         ostrack    240               1         2983   698130716
57         ostrack    80                1         2999   698130716
57         ostrack    80                1         2970   698130716
57         ostrack    80                1         3009   698130716
ostrack 池的总大小: 5400576 字节
按 apmh 池大小排序的内存块:
池标识     池名称   总大小(字节)            总计数         LOC    文件
70         apmh       40200                   2          121    2986298236
70         apmh       10016                   1          308    1586829889
70         apmh       6096                    2          4014   1312473490
70         apmh       2516                    1          294    1586829889
70         apmh       496                     1          2192   1953793439
70         apmh       360                     1          1024   3878879032
70         apmh       176                     1          1608   1953793439
70         apmh       152                     1          2623   1583816485
70         apmh       48                      1          914    1937674139
70         apmh       32                      1          1000   1937674139
apmh 池的总大小: 60092 字节
...
最后一部分输出是对整个 DBMS 集的内存使用者进行排序
DBMS 内存集中的所有内存使用者:
池标识    池名称              总大小(字节)字节 % 总计数     计数 %  LOC    文件
57         ostrack            5160048  71.90   1          0.07    3047   698130716
50         sqlch              778496   10.85   1          0.07    202    2576467555
50         sqlch              271784   3.79    1          0.07    260    2576467555
57         ostrack            240048   3.34    1          0.07    3034   698130716
50         sqlch              144464   2.01    1          0.07    217    2576467555
62         resynch            108864   1.52    1          0.07    127    1599127346
72         eduah              108048   1.51    1          0.07    174    4210081592
69         krcbh              73640    1.03    5          0.36    547    4210081592
50         sqlch              43752    0.61    1          0.07    274    2576467555
70         apmh               40200    0.56    2          0.14    121    2986298236
```

```
69        krcbh            32992    0.46   1        0.07   838   698130716
50        sqlch            31000    0.43   31       2.20   633   3966224537
50        sqlch            25456    0.35   31       2.20   930   3966224537
52        kerh             15376    0.21   1        0.07   157   1193352763
50        sqlch            14697    0.20   1        0.07   345   2576467555
...
在 UNIX 和 Linux 环境中，还可以报告私有内存的内存块。例如：
db2pd -memb pid=159770
专用内存集中的所有内存块：
地址                池标识     池名称     块存活时间  大小(字节)  I  LOC   文件
0x0000000110469068 88      private     1       2488      1  172   4283993058
0x0000000110469A48 88      private     2       1608      1  172   4283993058
0x000000011046A0A8 88      private     3       4928      1  172   4283993058
0x000000011046B408 88      private     4       7336      1  172   4283993058
0x000000011046D0C8 88      private     5       32        1  172   4283993058
0x000000011046D108 88      private     6       6728      1  172   4283993058
0x000000011046EB68 88      private     7       168       1  172   4283993058
0x000000011046EC28 88      private     8       24        1  172   4283993058
0x000000011046EC68 88      private     9       408       1  172   4283993058
0x000000011046EE28 88     private     10       1072      1  172   4283993058
0x000000011046F288 88     private     11       3464      1  172   4283993058
0x0000000110470028 88     private     12       80        1  172   4283993058
0x00000001104700A8 88     private     13       480       1  1534  862348285
0x00000001104702A8 88     private     14       480       1  1939  862348285
0x0000000110499FA8 88     private     80       65551     1  1779  4231792244
内存集总大小：94847 字节
按大小排序的内存块：
池标识    池名称     总大小(字节)    总计数      LOC      文件
88       private    65551        1          1779     4231792244
88       private    28336        12         172      4283993058
88       private    480          1          1939     862348285
88       private    480          1          1534     862348285
内存集总大小：94847 字节
```

-bufferpool

通过 db2pd –d dbname –buferpool 选项可以查看数据库当前的缓冲池大小以及当前的使用情况，如图 3-20 所示。

```
Bufferpools:
First Active Pool ID      1
Max Bufferpool ID         1
Max Bufferpool ID on Disk 1
Num Bufferpools           5

Address            Id   Name              PageSz  PA-NumPgs  BA-NumPgs  BlkSize  NumTbsp  PgsToRemov  CurrentSz  PostAlter  SuspndTSCt  Automatic
0x07000001A1D001A0 1    IBMDEFAULTBP      16384   66057      0          0        13       0           66057      66057      0           True
0x07000001A010B3A0 4096 IBMSYSTEMBP4K     4096    16         0          0        0        0           16         16         0           False
0x07000001A010C5E0 4097 IBMSYSTEMBP8K     8192    16         0          0        0        0           16         16         0           False
0x07000001A010D820 4098 IBMSYSTEMBP16K    16384   16         0          0        0        0           16         16         0           False
0x07000001A010EA60 4099 IBMSYSTEMBP32K    32768   16         0          0        0        0           16         16         0           False

Bufferpool Statistics for all bufferpools (when BUFFERPOOL monitor switch is ON):

BPID DatLRds    DatPRds    HitRatio TmpDatLRds TmpDatPRds HitRatio IdxLRds    IdxPRds    HitRatio TmpIdxLRds TmpIdxPRds HitRatio
1    305955242  2006       100.00%   722304496  0          100.00%   417307900  2440       100.00%   0          0          00.00%
4096 0          0          00.00%   0          0          00.00%   0          0          00.00%   0          0          00.00%
4097 0          0          00.00%   0          0          00.00%   0          0          00.00%   0          0          00.00%
4098 0          0          00.00%   0          0          00.00%   0          0          00.00%   0          0          00.00%
4099 0          0          00.00%   0          0          00.00%   0          0          00.00%   0          0          00.00%
```

图 3-20　使用–buferpool 选项

通过 pageSz * PA-NumPgs / 1024 = 16KB 内存页 * 66057 / 1024 = 1032 MB，计算得出当前数据库中 IBMDEFAULTBP 这个缓冲池的大小为 1032MB。

还可以通过这个命令选项查看到当前缓冲池的数据和索引缓冲命中率。

关于内存集(memset)、内存池(mempool)和内存块(memblock)的详细讲解，请读者参见《高级进阶 DB2(第 3 版)》一书。

3.6.2　db2mtrk 内存监控

db2mtrk 是用于在 DB2 数据库中进行内存跟踪的工具，可以用于查看实例、数据库、代理进程当前对内存的使用状态。

例 3-17　db2mtrk 示例 1。

```
 db2mtrk -i -d -v
 Memory for database:  SAMPLE
Backup/Restore/Util Heap is of size 16384 bytes
Package Cache is of size 81920 bytes
Catalog Cache Heap is of size 65536 bytes
Buffer Pool Heap is of size 4341760 bytes
Buffer Pool Heap is of size 655360 bytes
Buffer Pool Heap is of size 393216 bytes
Buffer Pool Heap is of size 262144 bytes
Buffer Pool Heap is of size 196608 bytes
Lock Manager Heap is of size 491520 bytes
Database Heap is of size 3637248 bytes
Other Memory is of size 16384 bytes
Application Control Heap is of size 327680 bytes
Application Group Shared Heap is of size 57344000 bytes
Total: 67829760 bytes
```

例 3-18 db2mtrk 示例 2。

```
$db2mtrk -i -d -p -r 1200
Tracking Memory on: 2012/10/10 at 17:05:24
Memory for instance
   other       fcmbp       monh
   58.4M       832.0K      6.8M
Memory for database: PQM
   utilh       pckcacheh   other       catcacheh   bph (1)     bph (S32K)
   576.0K      718.1M      192.0K      6.3M        18.4G       832.0K
   bph (S16K)  bph (S8K)   bph (S4K)   shsorth     lockh       dbh
   576.0K      448.0K      384.0K      338.2M      1.3G        70.4M
   apph (27870)apph (27866)apph (27864)apph (27789)apph (27788)apph (27787)
   64.0K       64.0K       128.0K      64.0K       64.0K       64.0K
   apph (27786)apph (27785)apph (27784)apph (27783)apph (27782)apph (27781)
   64.0K       64.0K       64.0K       64.0K       64.0K       64.0K
   apph (370)  apph (369)  apph (368)  appshrh
   64.0K       64.0K       64.0K       27.0M
Memory for agent 53378
   other
   192.0K
Memory for agent 82087
   other
   192.0K
```

db2mtrk 工具的语法如下：

- Usage: db2mtrk -i | -d | -a | -p [-m | -w] [-v] [-r interval [count]] [-h]db2mtrk -i #显示当前实例的内存使用情况
- db2mtrk -i -v #显示当前实例的内存使用情况的详细信息
- db2mtrk -d #显示数据库的内存使用情况
- db2mtrk -d -v #显示数据库的内存使用情况的详细信息
- db2mtrk -p #显示代理进程专用内存使用率
- db2mtrk -h #显示帮助信息

-m 参数选项用于显示最大的内存使用上线

-w 参数选项用于显示使用过程中内存达到的最大值——watermark

-r 参数选项用于重复显示，其中，interval 是重复显示的时间间隔数，count 是要重复显示的次数

interval 指定以秒为单位的重复间隔

count 指定间隔的次数

-v 详细输出
-h 显示帮助信息
例 3-18 显示了实例级别和数据库级别内存使用情况的详细信息。
在 DB2 V9.1 中对该命令的-d 选项和-i 选项做了以下更改：

- 在 Windows 平台上，现在支持用于显示数据库级别内存的-d 选项。
- 由于现在可通过-d 选项来显示数据库级别内存，因此-i 选项仅用于显示实例级别内存。

在 db2mtrk 工具的输出信息中有下面几种类型的信息：

- 当前值的大小
- 最大值的限制(hard limit)
- 最高值(high water mark)
- 类型(用于指定使用了哪种内存)
- 代理进程使用的内容(只针对私有内存池)

提示：

在我们使用 db2mtrk 工具时，主要查看高水位和实际的配置之间是否接近。例如，如果 util_heap_sz 的高水位逼近了实际的数据库配置参数指定的值，那说明您可以考虑增加该参数。当然，有的时候我们还可能看到高水平的值超过了实际配置的值，这主要是因为 DB2 部分由数据库配置参数设置的值只是软限制而不是硬限制。

3.7　本章小结

本章我们给大家详细讲解了一些监控案例，希望大家能够从这些案例中得到一些启发。其实本章所讲的大部分案例都是实际生产中经常碰到的案例。

在实际的生产环境下，在怀疑数据库中可能存在着性能问题的时候，就可以选择合适的监控工具和手段来发现数据库中可能存在的瓶颈或不合理的资源使用。在实际发生性能问题的时候，问题的产生原因可能并不单一，而是由很复杂的原因造成的，所以往往需要通过多方面的监控分析才能得到准确的结论。所以本章介绍的这些工具经常需要以多种方式结合起来使用，然后将监控结果综合地进行分析，从而才能得到最准确的分析结论。

第 4 章

DB2 配置参数调整

在 DB2 数据库中，有实例级参数(dbm cfg)、数据库级参数(db cfg)和 DB2 注册变量(db2set -lr)三种配置参数。DB2 利用这三种配置参数来进行数据库约束和资源限制。

DB2 配置参数对数据库来说非常重要，通过对这些参数合理地进行调整，可以极大地提高数据库性能。这三种配置参数里面有几百个具体参数，本章只讲解那些对数据库性能影响最大的参数。

本章主要讲解如下内容：

- 初识 DB2 配置参数
- 监控和调优实例级参数
- 监控和调优数据库级参数
- 调整 DB2 注册变量
- 内存自动调整

4.1 初识 DB2 配置参数

实例级参数、数据库级参数和 DB2 注册变量

DB2 有几百个配置参数。其中很多参数都是由 DB2 自动配置的，而其他一些参数则都有各自的默认值，这些默认值都被证明在大多数环境中能够发挥得很好。接下来，我们

只描述那些常常需要另外进行配置的参数。

实例级参数相当于全局变量，作用于这个实例下的所有数据库，有些实例级(数据库管理器)参数可以在线更改(立即生效)，而另一些参数则要重启实例后才能生效(即 db2stop 之后接着 db2start)。数据库级参数类似于局部变量，只作用于当前数据库，有些数据库级参数的更改可以立即生效，而另一些则要求先停止数据库，再重新激活数据库。每种配置参数的文档中都规定了参数是否可以在线配置。DB2 注册变量也分作用域，比如全局级、实例级等等。

实例级参数、数据库级参数和 DB2 注册变量的基本管理命令如表 4-1 所示。

表 4-1　实例级、数据库级参数和 DB2 注册变量的基本管理命令

命　令	描　述
get dbm cfg <show detail>	显示实例级参数<详细显示>
update dbm cfg using param_name param_value	修改实例级参数
get db cfg for db_name <show detail>	显示指定的数据库级参数<详细显示>
update db cfg for db_name using param_name param_value	修改数据库级参数
db2set -lr	显示 DB2 中已经设置的所有注册变量
db2set param_name=param_value	设置 DB2 注册变量

当您修改了实例级参数后，就可以用下面的 DB2 CLP 命令来查看该设置是否立即生效(在线)：

```
db2 attach to db2inst1
db2 get dbm cfg show detail
```

如果修改了数据库级参数，可以用如下命令进行查看：

```
db2 get db cfg for dbname show detail
```

例如，在接下来的情况中，max_querydegree 和 JAVA_HEAP_SZ 分别增加到了 3 和 3096。如果参数是在线配置的，那么 Delayed Value 和 Current Value 应该是一样的。否则，实例级参数就需要重新启动实例，数据库级参数就要重新激活数据库。

例 4-1　Show Details 实例。

```
$db2 get dbm cfg show detail
Database Manager Configuration
Node type = Enterprise Server Edition with local and remote clients
```

```
Description                Parameter        Current Value       Delayed Value
--------------------------------------------------------------------------------
Maximum query degree of parallelism  (max_querydegree) = 3           3
Java Virtual Machine heap size (4KB)  (JAVA_HEAP_SZ) = 2048         3096
```

在上面的例子中，JAVA_HEAP_SZ 修改之后的值并未生效，这就需要重启实例，而 max_querydegree 修改之后的值已经生效了。

4.2　监控和调优实例级(DBM)配置参数

在以下要讲的配置参数中，有些是从共享内存中分配空间的，所以应该记住操作系统内存的限制，您必须确保没有过度分配内存。如果过度分配内存，将会导致操作系统的内存不足而发生换页(paging)，这对于性能来说后果将是灾难性的。

4.2.1　代理程序相关配置参数

- max_coordagents 参数表明用来接收应用程序请求的代理程序的最大数目。max_coordagents 的值应当至少是实例下所有数据库中 maxappls(单一数据库允许的并发应用程序最大数目)值的总和。如果数据库的数量大于 numdb 参数，那么最安全的方案就是使用 numdb 和 maxappls 最大值的乘积。每个额外的代理程序都需要一些资源开销，这些开销在启动数据库管理器时会分配给代理程序。在内存受限(memory constrained)的环境中，这个参数对于限制数据库管理器的总内存使用量很有用，因为每个附加的代理都需要额外的内存。
- num_poolagents 参数指定您希望代理程序池里保存多少代理程序。如果创建的代理程序多于该参数值所指明的数目，那么代理程序执行完自己当前的请求后将被销毁而不是返回给代理程序池。如果该参数的值为 0，将不保存任何代理程序，在需要的时候才创建，在代理程序执行完自己当前的请求后就销毁。要避免因在并发连接许多应用程序的 OLTP 环境中频繁创建和销毁代理程序而产生的成本，请将 num_poolagents 的值增加到接近 max_coordagents 的值。
- num_initagents 参数决定代理程序的初始数量，这些代理程序是在 db2start 时在代理程序池中创建的。指定的初始代理程序数目要合适(尽管并非必要条件)，这可以缩短“热身”时间。num_initagents 指定在 db2start 时的代理池中创建的空闲代理的数量，可以帮助加快在开始使用数据库时的连接。

1. 如何更改参数

为了更改这些参数，请运行以下命令：

```
db2 -v update dbm cfg using max_coordagents a_value
db2 -v update dbm cfg using num_poolagents b_value
db2 -v update dbm cfg using num_initagents c_value
db2 -v terminate
```

2. 调优步骤

在运行期间的任何时候，您都可以使用下面这个命令来获取实例的快照数据：

```
db2 -v get snapshot for dbm
```

看一下下列输出行：

```
High water mark for agents registered = 4
High water mark for agents waiting for a token = 0
Agents registered = 4
Agents waiting for a token = 0
Idle agents = 0
Agents assigned from pool = 5
Agents created from empty pool = 4
Agents stolen from another application = 0
High water mark for coordinating agents = 4
Max agents overflow = 0
```

“Idle agents”显示了在代理池中空闲代理的数量，而“Agents assigned from pool”则显示了从代理池中将代理分配出去的次数。“Agents created from empty pool”显示了在空池情况下必须创建的代理的数量，包括 num_initagents。如果“Agents created from empty pool”/“Agents assigned from pool”的比例较高(5:1 或更大)，那么可能表明应该增加 num_poolagents。这还可能表明系统的总体工作负载太高。此时可以通过降低 max_coordagents 来调整工作负载。如果这个比例较低，那么暗示着 num_poolagents 可能被设得太高，有些代理就会浪费系统资源。如果您发现“Agents waiting for a token”或“Agents stolen from another application”不等于 0，那么可能需要增加 max_coordagents 以允许数据库管理器可以使用更多的代理程序。

如果“Agents stolen from another application”大于 0，并且机器的内存资源是充足的，那么可以增加 max_coordagents。此外，“Local connections”+“Remote connections to db manager”将指出连接到实例的并发连接的数量。“High water mark for agents registered”将报告在某一次连接到数据库管理器的代理曾出现的最大数量。“Max agents overflow”

报告当已经达到 max_coordagents 时，收到的创建新代理的请求的次数。最后，“Agents Registered”显示在被监控的数据库管理器实例中当前注册的代理的数量。

3. 调优建议

在大多数情况下，将 max_coordagents 和 num_poolagents 的值设置成略微大于并发应用程序连接的最大预计数目。让 num_initagents 保留为默认值会比较好。

num_initagents 和 num_poolagents 应该设置为预期的并发实例级连接的平均数量，对于 OLAP 这个值通常比较低，而对于 OLTP 就要高一些。对于存在大量 ramp up 连接情况下的性能基准，将 num_initagents 设置成预期的连接数量(这将减少资源争用，从而显著地减少 ramp up 连接所需的时间)。在使用了连接池的 3 层环境中，num_initagents 和 num_poolagents 对性能的影响很小，因为即使在应用程序空闲的时候，应用服务器也会连续不断地维护连接。

DB2 的这两个参数 max_connections 和 max_coordagents 都可以被设置成 automatic。如果您认为系统可以承受所有的连接，同时又想限制被协调代理消耗的资源，那么可以只将 max_connections 设定为 automatic，而将 max_coordagents 设定为数值。这样系统会认为可以连到实例的连接数是无限的。如果您对最大连接数和协调代理数都不想进行限制的话，可以将它们都设定为 automatic。

4.2.2　sheapthres

排序堆阀值(sheapthres)是对专用排序在任何给定时间可以使用的总内存量的实例范围软限制。当某个实例使用的专用排序内存总量达到此限制时，为其他传入专用排序请求分配的内存将显著减少。sheapthres 参数在 DB2 V9.5 及以后的版本中不再适用，因为绝大多数的排序都成为共享内存的排序，所以建议此值为 0。仅当 sheapthres 设置为 0 时，才允许自动调整 sheapthres_shr 或 sortheap。

排序堆阀值(sheapthres)和共享排序的排序堆阀值(sheapthres_shr)能控制可用于整个实例中所有排序的内存总量。在 4.3.4 节中，我们将更加详细地描述影响排序性能的参数。

4.2.3　fcm_num_buffers

fcm_num_buffers 只在有多个逻辑分区的 DPF 环境中使用，指定用于内部通信的大小为 4 KB 的缓冲区的数量。如果没有使用 DPF，那么这个值甚至不会出现在快照输出中。此外，该信息将来自其上运行了快照的分区。

```
Node FCM information corresponds to          = 2
Free FCM buffers                             = 10719
```

```
Total FCM buffers                          = 10740
Free FCM buffers low water mark            = 10660
Maximum number of FCM buffers              = 2097880
Free FCM channels                          = 6254
Total FCM channels                         = 6265
Free FCM channels low water mark           = 6066
Maximum number of FCM channels             = 2097880
Number of FCM nodes                        =9
```

例如，在 DBM 快照之前的那个快照中，**“Node FCM information corresponds to”**显示了值 2，因此它是从 2 号分区那里得来的。“Get snapshot for dbm global”可用于获得所有分区值的群集。

DBM 快照的 FCM Node 部分可用于查看主要的分区间通信发生的地点，进而用于调查。如果通信量很大，就表明需要更多的 FCM 缓冲区内存，需要不同的分区键，或者需要不同的表来分派表空间。如果**“Free FCM buffers low water mark”**小于 fcm_num_buffers 的 15%，那么可以增加 fcm_num_buffers 的大小，直到**“Free FCM buffers low water mark”**大于或等于 fcm_num_buffers 的 15%，以确保总有足够的 FCM 资源可供使用。

4.2.4 sheapthres_shr

这是对实例中并发共享的排序可以消耗的内存总量的硬性限制。当 sheapthres 大于 0 时，sheapthres_shr 这个值只有在以下情况下才适用：intra_parallel=yes，这是因为当 intra_parallel 设置为 no 时，没有任何共享排序；或者 Concentrator 在(max_connections > max_coordagents)范围内。对于在 with hold 选项下使用游标的排序，将从共享内存中为其分配内存。

“Shared Sort heap high water mark”显示最多一次分配了的共享排序内存。如果这个值总是远远低于 sheapthres_shr，那么应该减少 sheapthres_shr，以便为其他数据库组件节省内存。如果这个值刚好接近于 sheapthres_shr，那么可能需要增加 sheapthres_shr。**“Total Shared Sort heap allocated”**是从排序堆空间中为所有排序分配的总页数。如果元素值大于或等于 sheapthres_shr，那么意味着这些排序没有得到由 sortheap 参数定义的完整排序堆空间。增加 sheapthres_shr 的大小以帮助避免这种情况发生。

在设置时，应尽量使其为 sortheap 的倍数。

4.2.5 intra_parallel

该参数指定数据库管理器是否可以使用内部分区并行性(intra-partition parallelism)。默

认值 no 对于并发连接较多的情况(主要是 OLTP)最好，而 YES 对于并发连接较少的情况以及复杂 SQL (OLAP/DSS)来说最好。混合的工作负载通常可以得益于 NO。

当启用该参数时，就会导致从共享内存中分配排序内存。此外，如果并发程度显著增加，那么还可能导致过多的系统开销。如果系统是非 OLTP 的，那么 CPU 数与分区数的比例是 4:1，而 CPU 负载运行的平均百分比是 50%，intra_parallel 很可能会提高性能。

4.2.6　mon_heap_sz

这是为数据库系统监视器(system monitor)数据分配的内存数量。当执行诸如快照监视或激活事件监视器之类的数据库监控活动时，就要从监视器堆中分配内存。如果没有足够的可用内存，并且 DB2 返回错误，那么可以试着将这个值设为 1024。如果还是遇到错误，一次一次地增加，每次增加 256，直到错误消失。

4.3　监控和调优数据库级配置参数

下面主要讲解对数据库性能影响最重要的一些配置参数。

这里描述的配置参数包括：

- 缓冲池大小
- 日志缓冲区大小
- 应用程序堆大小
- 排序堆大小和排序堆阈值
- 代理程序的数目
- 锁相关配置参数
- 活动应用程序的最大数目
- 异步页清除程序的数目
- I/O 服务器的数目

4.3.1　缓冲池大小

缓冲池是内存中的一块区域，用于临时读入和更改数据库页(包含表行和索引项)。缓冲池的用途是为了提高数据库系统的性能。从内存访问数据要比从磁盘访问数据快得多，因此，数据库管理器需要从磁盘读取或写入磁盘的次数越少，性能就越好。对一个或多个缓冲池进行配置之所以是调优的最重要方面，在于连接至数据库的应用程序的大多数数据

(不包括大对象和长字段数据)操作都是在缓冲池中进行的。

默认情况下，应用程序使用缓冲池 ibmdefaultbp，它是在数据库创建时创建的，大小是 automatic。

1. 建议

一定要在 instance_memory 或 database_memory 受限的情况下才允许将缓冲池设定为 automatic，否则强烈建议设置成固定值。

- 一开始，如果您的机器上有足够大的内存，请将 bufferpool 的 size 设置成 65536 个页(256 MB)，或者等于机器总内存的 10%。
- 对于大型 OLTP 数据库，在保持系统稳定的同时为缓冲池留出尽可能多的内存。一开始，先尝试使用 4GB(请结合自己的内存资源，此处仅作为参考)的内存，然后尝试用更多内存。

对于大型数据库，除了默认的缓冲池 ibmdefaultbp，我们还可以为多个页大小的表空间创建多个缓冲池，每个缓冲池的大小取决于应用程序和表空间大小。不同缓冲池的创建或调整请考虑下列情况：

- 可以把临时表空间分配给单独的缓冲池，以便为需要临时存储器的查询，尤其是执行大量排序的查询提供更佳性能。
- 创建不同的缓冲池，把频繁更新的表和索引放在单独的缓冲池中，与其他频繁查询但不频繁更新的表和索引分开。这样两组不同类型的应用程序可降低相互影响的可能性。
- 如果创建或调整某个缓冲池的大小过大，在数据库启动时内存不可用，那么数据库管理器将使用最小大小为 16 页的系统缓冲池(每个页大小对应一个系统缓冲池)启动，并返回 SQL1478W 警告。为了避免仅使用系统缓冲池启动数据库，可用考虑使用 db2_overrid_bpf 注册变量来限制缓冲池所需的内存。

2. 缓冲池大小的设置

通过使用“get snapshot for bufferpools on **dbname**”，可以为数据库中的每个缓冲池生成快照，下面的代码展示了这样一个快照：

```
Bufferpool Snapshot
Bufferpool name                              = IBMDEFAULTBP
Database name                                = BASDB
Database path                          = /db2/bamsdb/db2bas/NODE0000/SQL00001/
Input database alias                          = BASDB
```

```
Snapshot timestamp                          = 10/17/2012 09:37:23.223386
Buffer pool data logical reads              = 12795178
Buffer pool data physical reads             = 26457
Buffer pool temporary data logical reads    = 737441
Buffer pool temporary data physical reads   = 0
Buffer pool data writes                     = 270
Buffer pool index logical reads             = 5311524
Buffer pool index physical reads            = 29558
Buffer pool temporary index logical reads   = 0
Buffer pool temporary index physical reads = 0
Buffer pool xda logical reads               = 0
Buffer pool xda physical reads              = 0
Buffer pool temporary xda logical reads     = 0
Buffer pool temporary xda physical reads    = 0
Buffer pool xda writes                      = 0
Total buffer pool read time (milliseconds) = 8727
Total buffer pool write time (milliseconds)= 225
Asynchronous pool data page reads           = 2032
Asynchronous pool data page writes          = 263
Buffer pool index writes                    = 71
Asynchronous pool index page reads          = 386
Asynchronous pool index page writes         = 71
Asynchronous pool xda page reads            = 0
Asynchronous pool xda page writes           = 0
Total elapsed asynchronous read time        = 2395
Total elapsed asynchronous write time       = 210
Asynchronous data read requests             = 559
Asynchronous index read requests            = 151
Asynchronous xda read requests              = 0
No victim buffers available                 = 0
Direct reads                                = 4227718
Direct writes                               = 708166
Direct read requests                        = 316462
Direct write requests                       = 353523
Direct reads elapsed time (ms)              = 258876
Direct write elapsed time (ms)              = 3226
Database files closed                       = 0
```

```
    Unread prefetch pages                      = 0
    Vectored IOs                              = 710
    Pages from vectored IOs                    = 2418
    Block IOs                                = 0
    Pages from block IOs                       = 0
    Node number                               = 0
    Tablespaces using bufferpool                 = 6
    Alter bufferpool information:
    Pages left to remove                       = 0
     Current size                             = 309024
     Post-alter size                           = 309024
```

为了判断缓冲池的效率，需要计算缓冲池命中率(BPHR，Buffer Pool Hit Ratio)。您所需的重要信息在上面已经用粗体标出来了。理想的 BPHR 在某些地方应超过 95%。计算公式如下：

```
    BPHR (%) = (1 - (("Buffer pool data physical reads" + "Buffer pool index
physical reads") /("Buffer pool data logical reads" + "Buffer pool index logical
reads"))) * 100
```

在 ibmdefaultbp 缓冲池的以上快照中，我们可以这样来计算 bphr：

```
    HPHR(%) = (1-((26457 + 29558) / (12795178 + 5311524))) * 100
            = (1-(56015 / 18106702)) * 100
            = (1- 0.003093) * 100
            = 99.69
```

在这种情况下，BPHR 约等于 99.69%。当前，缓冲池是 309024 * 4KB(4.7GB)。如果 BPHR 值低于 95%，试着增加该缓冲池的大小，看看 BPHR 是否会随之增加，这样做是值得的。如果 BPHR 还是比较低，那么可能就需要重新设计逻辑布局了。

3. 基于块的缓冲池的效率

如果是基于块的缓冲池，并且看到“Block Ios”的值较低，那么应考虑修改缓冲池，增加 numblockpages 的大小。如果这时看到“Block Ios”的值更大了，可以考虑将 numblockpages 再增大一些。如果适得其反，就应减小 numblockpages 的大小。

4. 如何更改该参数

运行下面这段代码，以便：

- 验证 syscat.bufferpools 中 bufferpool 的大小
- 修改 bufferpool 的大小

```
db2 -v connect to sample
db2 -v "select * from syscat.bufferpools"
db2 -v "alter bufferpool ibmdefaultbp size bigger_value"
db2 -v terminate
```

要确定缓冲池大小是否足够大，请在运行应用程序时收集数据库和/或缓冲池的快照。类似于如下代码为您提供的信息：

```
db2 -v update monitor switches using bufferpool on
db2 -v get monitor switches
db2 -v reset monitor all
-- run your application --
db2 -v get snapshot for all databases > snap.out
db2 -v get snapshot for dbm >> snap.out
db2 -v get snapshot for all bufferpools >> snap.out
db2 -v reset monitor all
db2 -v terminate
```

确保在断开数据库连接之前发出“db2 -v get snapshot”命令。当最后一个应用程序与数据库断开连接时，该数据库停止运行，同时所有快照统计信息将会丢失。要确保一直存在使数据库处于正常运行状态的连接，请使用下列方法之一：

- 在收集快照的窗口中保持单独的连接。
- 使用 db2 activate database 命令。

在数据库快照或缓冲池快照的快照输出中，查找“logical reads”和“physical reads”以便计算出缓冲池命中率，这可以帮助您调优缓冲池：

```
Buffer pool data logical reads = 702033
Buffer pool data physical reads = 0
Buffer pool data writes = 414
Buffer pool index logical reads = 168255
Buffer pool index physical reads = 0
```

缓冲池的命中率表明数据库管理器不需要从磁盘装入页(该页已经在缓冲池中)就能处理页请求的时间百分比。缓冲池的命中率越高，使用磁盘 I/O 的频率就越低。

注意：
以下示例的输出取决于你的缓冲池监控开关是否打开。

可以使用 sysibmadm.bp_hitratio，也可以使用以下 SQL 语句来获得缓冲池命中率：

```
db2 "select snapshot_timestamp, substr(db_name,1,10) as dbname,
substr(bp_name,1,18) as bufferpool, total_hit_ratio_percent as total,
data_hit_ratio_percent as data, index_hit_ratio_percent as index
from sysibmadm.bp_hitratio"
```

上述计算考虑了缓冲池高速缓存的所有页(索引和数据)。理想情况下，该比率应当超过 96%，并尽可能接近 100%。要提高缓冲池命中率，请尝试下面这些方法：

- 增加缓冲池大小。
- 考虑分配多个缓冲池，如果可能的话，为每个经常被访问的大表所属的表空间分配缓冲池，为一组小表分配缓冲池，然后尝试使用不同大小的缓冲池查看哪种组合会提供最佳性能。

如果已分配的内存不能帮助提高性能，那么请避免给缓冲池分配过多的内存。应当根据取自测试环境的快照信息来决定缓冲池的大小。

4.3.2 日志缓冲区大小(logbufsz)

logbufsz 是数据库配置参数，是用于日志缓冲区的参数。这个参数指定将日志记录写到磁盘之前的缓冲区的数据库堆(dbheap)的数量。当提交事务或者日志缓冲区已满时，就要将日志记录写入磁盘。对日志记录进行缓冲将导致将日志记录写入磁盘的活动不再那么频繁，但每次要写的日志记录会更多。当下列事件之一发生时会将日志记录写入磁盘：

- 一个事务提交。
- 日志缓冲区已满。
- 发生了其他某些内部数据库管理器事件。

将日志记录存到缓冲区将产生更加有效的日志文件 I/O，这是因为这样一来可以降低将日志记录写入磁盘的频率，同时每次可写更多的日志记录。如果对专用的日志磁盘有相当多的读操作，或者希望有较高的磁盘利用率，那么可以增加这个缓冲区的大小。当增加这个参数的值时，也要考虑 dbheap 参数，因为日志缓冲区使用的空间由 dbheap 参数控制。此数据库配置参数的默认值为 automatic，这表示数据库堆可以根据需要增大，直到达到 database_memory 限制或达到 instance_memory 限制。

1. 如何更改该参数

我们发现该参数的默认值为 256(4KB 页)，这对于 OLTP 数据库而言通常不够大。logbufsz 的最佳值为 512 个 4KB 页或者更大。例如，可以使用下面的命令来更改该参数的值：

```
db2 -v update db cfg for DB_NAME using LOGBUFSZ 512
db2 -v terminate
```

2. 调优步骤

通过查看下面代码中的各行，使用数据库快照(db2 get snapshot for db on db_name)来确定 LOGBUFSZ 参数的值是否为最佳值：

```
Log pages read = 0
Log pages written = 414105
```

对于 OLTP，一开始以至少 512 页为佳；对于 OLAP，则以 256 页为佳。如果常常看到“Log pages read”大于 0，那么可能需要增加这个值。如果发生回滚，也可能要读取日志页。一般而言，“log pages read”和“log pages written”之比应当尽可能小。理想情况下，“log pages read”的值应为 0，而“log pages written”的值应很大。当“log pages read”太多时，意味着需要较大的 logbufsz。

如果在试图增加 logbufsz 时收到错误，那么可以按相同数量增加 dbheap，然后再次尝试。

4.3.3 应用程序堆大小(applheapsz)

applheapsz 是数据库级配置参数，应用程序堆是供数据库管理器代表某个特定代理使用的私有内存。当代理或子代理要为应用程序初始化时，就要从这个堆中分配内存，并且分配的内存大小是处理请求时所需的最小内存量。如果需要更多的内存，那么最多可以从堆中分配由参数 applheapsz 指定的最大值那么多的内存。按 256 逐次增加，直到错误消失。applheapsz 定义了代表某个特定代理程序或子代理程序的数据库管理器可以使用的私有内存页数。在为应用程序初始化代理程序或子代理程序时分配堆。分配的堆的大小是处理代理程序或子代理程序请求所需的最小值。当代理程序或子代理程序需要更多的堆空间以处理较大的 SQL 语句时，数据库管理器将按照需要分配内存，所分配的内存大小最大可达到该参数指定的最大值。

applheapsz 指的是整个应用程序可以消耗的应用程序内存的总量。对于 DPF、集中器或 SMP 配置，这表示在类似工作负载下，除非使用了 automatic 设置，否则可能需要增大在先前发行版中使用的 applheapsz 值。

1. 如何更改该参数

使用如下命令可以将该参数从默认值更改成最佳值：

```
db2 -v update db cfg for db_name using applheapsz 256
db2 -v terminate
```

2. 调优步骤

当应用程序接收到表明应用程序堆中存储空间不够的错误时，应该增加 applheapsz 的值。

提示：

在多层架构的环境中，应用是通过连接池连接数据库的，建议合理地增大 applheapsz 以提高连接池的处理效率。

4.3.4 sortheap 和 sheapthres_shr

sortheap 是数据库级配置参数，它定义了用于私有排序的专用内存页的最大数目或要用于共享排序的共享内存页的最大数目。每个排序都有独立的排序堆，这是由数据库管理器在需要的时候分配的。通常，大家都能知道，当排序所需的内存量超过了 sortheap 时，就会发生排序溢出；但也有可能没有注意到的是，如果统计信息已过时，或者数据有偏差，并且没有收集到发布的统计信息，此时一旦 DB2 请求太小的堆，而实际的排序操作超出了所请求的量，也会发生溢出。因此，使统计信息保持最新十分重要。此外，应确保排序不是某个丢失的索引的结果。如果排序是私有排序，那么该参数会影响代理程序的私有内存；如果排序是共享排序，那么该参数将影响数据库的共享内存。每个排序都有单独的由数据库管理器按需分配的排序堆。在排序堆中对数据进行排序。如果由优化器来指导排序堆大小的分配，那么用优化器提供的信息来分配的排序堆的大小要小于由该参数指定的排序堆大小。

sheapthres 也是数据库级配置参数。私有排序和共享排序所使用内存的来源不一样。共享排序内存区的大小是在第一次连接到数据库时根据 sheapthres 值以静态方式预先确定的。私有排序内存区的大小则是不受限制的。对于私有排序和共享排序，应用 sheapthres 参数的方式也不同：

- 对于私有排序，sheapthres 是对私有排序在任何给定的时间可以消耗的全部内存的实例级“软”限制。当实例的总私有排序内存消耗量达到这一限制时，为其他进入的私有排序请求而分配的内存会大大减少。

- 对于共享排序，sheapthres 是对共享排序在任何给定的时间可以消耗的全部内存的数据库级“硬”限制。当达到这一限制时，不允许有其他共享排序内存请求，直到总的共享内存消耗量回落到 sheapthres 指定的限制之下。

使用排序堆的操作示例包括内存中表的哈希连接和操作。sheapthres 的显式定义可以防止数据库管理器将过多数量的内存用于大量排序。

在 4.2.2 节中已提到，sheapthres 参数已经不再适用，建议设置 sheapthres 为 0。另外通过数据库级参数 sheapthres_rsh，对排序内存使用者每次可使用的数据库共享内存总量进行软限制。

1. 建议

- 使用数据库系统监视器来跟踪排序活动。
- 使用合适的索引使排序堆的使用降到最低。
- 当需要频繁进行大型排序时，增加 sortheap 的值。
- 如果增加 sortheap，请确定是否还需要调整数据库管理器配置文件中的 sheapthres_shr 参数。
- 优化器使用排序堆大小来确定存取路径。在更改该参数后请考虑重新绑定应用程序(使用 REBIND PACKAGE 命令)。
- 理想情况下，应当将 sheapthres_shr 合理地设置为在数据库中设置的 sortheap 参数最大值的倍数。该参数至少应当是实例中任何数据库定义的最大 sortheap 的两倍。

2. 如何更改这些参数

要更改 sortheap 和 sheapthres_shr 的值，请运行以下命令：

```
--禁用 sortheap 配置参数自调整功能--
db2 -v update db cfg for DB NAME using sortheap manual
-- 禁用 sortheap 配置参数自调整功能,同时将 sortheap 当前值改为 4094--
db2 -v update db cfg for DB_NAME using sortheap 4096
-- sheapthres_shr 是数据库级参数--
db2 -v update db cfg for DB_NAME using sheapthres_shr 131072
db2 -v terminate
```

3. 调优步骤

对于 OLTP 应用程序，不应该执行大型排序。大型排序在 CPU 和 I/O 资源方面的消耗成本太高了。通常，sortheap 的默认值 automatic 就足够了。事实上，对于高并发性 OLTP，

您可能希望降低这个默认值。当需要进一步研究时，可以发出下面这条命令：

```
db2 -v update monitor switches using sort on
```

然后，让您的应用程序运行一会儿，然后运行：

```
db2 -v get snapshot for database on sample
```

查看如下输出结果：

```
Total Private Sort heap allocated        = 0
Total Shared Sort heap allocated         = 158
Shared Sort heap high water mark         = 410233
Post threshold sorts (shared memory)      = 0
Total sorts                          = 1596191
Total sort time (ms)                    = 147423
Sort overflows                        = 742
Active sorts                         = 1
Commit statements attempted               = 298304
Rollback statements attempted             = 2071
Dynamic statements attempted              = 2915297
Static statements attempted               = 12615722
Failed statement operations               = 3133
```

根据该输出，可以计算每个事务的排序数目，并计算溢出了可用于排序的内存的那部分排序的百分比。

```
SortsPerTransaction =  (Total Sorts) / (Commit statements attempted +
                      Rollback statements attempted)
PercentSortOverflow = (Sort overflows * 100 ) / (Total sorts)
```

经验：如果 SortsPerTransaction 大于 5，那么可能表明每个事务的排序太多。如果 PercentSortOverflow 大于 3%，那么可能发生了严重的、未曾预料到的大型排序。发生这种情况时，增加 sortheap 只会隐藏性能问题，却无法修正。这个问题的正确解决方案是通过添加正确的索引来改进有问题的 SQL 语句的存取方案。对于 OLTP，一开始最好是设为 32768；对于 OLAP，则设置在 32768 到 65536 之间。如果有很多的“Sort overflows”(两位数)，那么很可能需要增加 sortheap。如果“Number of hash join overflows”不为 0，就按照 8192 逐次增加 sortheap，直到为 0。如果“Number of small hash join overflows”不为 0，就按 10%的比例逐次增加 sortheap，直到小散列连接溢出数为 0。

4.3.5　锁相关配置参数

以下这些与锁相关的控制参数都是数据库级配置参数：

- locklist 表明分配给锁列表的存储容量。每个数据库都有锁列表，锁列表包含了并发连接到数据库的所有应用程序持有的锁。“锁定”是数据库管理器用来控制多个应用程序并发访问数据库中数据的机制。行和表都可以被锁定。
- maxlocks 定义了单个应用程序持有的锁列表的百分比上限，在数据库管理器执行锁升级之前必须填充锁列表。当应用程序使用的锁列表百分比达到 maxlocks 时，数据库管理器会升级这些锁，这意味着用表锁代替行锁，从而减少锁列表中锁的数量。当任何应用程序持有的锁数量达到整个锁列表大小的这个百分比时，对应用程序持有的锁进行锁升级。如果锁列表用完了空间，那么也会发生锁升级。数据库管理器通过查看应用程序的锁列表并查找行锁最多的表，决定对哪些锁进行升级。如果用表锁替换这些行锁，将不再会超出 maxlocks 值，那么锁升级就会停止。否则，锁升级就会一直进行，直到持有的锁列表百分比低于 maxlocks。maxlocks 参数乘以 maxappls 参数的值不能小于 100。

注意：

虽然锁升级过程本身并不用花很多时间，但是锁定整个表(相对于锁定个别行)降低了并发性，而且数据库的整体性能可能会由于对受锁升级影响的表的后续访问而降低。

使用下列步骤确定所需的锁列表：每个数据库都有锁列表，锁列表包含所有同时连接到数据库的应用程序持有的锁。在 32 位平台上，对象上的第一个锁要求占 64 字节，而其他的锁要求占 32 字节。在 64 位平台上，第一个锁要求占 112 字节，而其他的锁要求占 56 字节。

注意：

也可以这样理解，S 锁占 32 或 56 字节；X 锁占 64 或 112 字节。

当应用程序使用的 locklist 的百分比达到 maxlocks 时，数据库管理器将执行一次锁升级(lock escalation)，在这个过程中会将行锁换成单独的表锁。而且，如果 locklist 快要耗尽，数据库管理器将找出持有表上行锁最多的连接，将这些行锁换成表锁以释放 locklist 内存。锁定整个表可以大大减少并发性，但死锁和锁等待的几率也增加了。

(1) 计算锁列表大小的下限：(512 * 32 * maxappls) / 4096。其中，512 是每个应用程序平均所含锁数量的估计值，32 是对象(已有一把锁)上每把锁所需的字节数。

(2) 计算锁列表大小的上限：(512 * 64 * maxappls)/ 4096。其中，64 是某个对象上第一把锁所需的字节数。

(3) 对于您的数据，估计可能具有的并发数，并根据您的预计结果为锁列表选择初始值，该值位于您计算出的上限和下限之间。

使用数据库系统监视器调优 maxlocks。

设置 maxlocks 时，请考虑锁列表的大小(locklist)：

```
maxlocks = 100 * (512 锁 / 应用程序 * 32 字节 / 锁 * 2) / (locklist * 4096 字节)
```

上述公式允许任何应用程序持有的锁是平均数的两倍。如果只有几个应用程序并发地运行，可以增大 maxlocks，因为在这种条件下锁列表空间中不会有太多争用。

locktimeout 指定了应用程序为获取锁所等待的秒数，这有助于应用程序避免全局死锁。

- 如果将该参数设置成 0，那么应用程序将不等待获取锁。在这种情形中，如果请求时没有可用的锁，那么应用程序立刻会接收到 - 911 错误。
- 如果将该参数设置成 - 1，那么将关闭锁超时检测。在这种情形中，应用程序将等待获取锁(如果请求时没有可用的锁)，一直到被授予了锁或出现死锁为止。

1. 建议

设置 locktimeout 以快速检测由于异常情形而出现的等待，例如事务被延迟了(可能是由于用户离开了他们的工作站)。将它设置得足够高，这样有效的锁请求就不会因为高峰时的工作负载而超时，在高峰时等待获取锁的时间将延长。

在联机事务处理(OLTP)环境中，这个值从 30 秒开始。在只进行查询的环境中则可以从一个更大的值开始。无论哪种情况，都可以使用基准测试技术来调优该参数。

2. 如何更改这些参数

要更改锁参数，请运行以下命令：

```
db2 -v update db cfg for DB_NAME using locklist a_number
db2 -v update db cfg for DB_NAME using maxlocks b_number
db2 -v update db cfg for DB_NAME using locktimeout c_number
db2 -v terminate
```

3. 监控步骤

一旦锁列表满，由于锁升级会生成更多的表锁和更少的行锁，因此减少了数据库中共享对象的并发性，从而降低了性能。另外，应用程序间可能会发生更多死锁(因为它们都等待数量有限的表锁)，这会导致事务被回滚。当数据库的锁请求达到最大值时，应用程序将接收到值为 - 912 的 SQLCODE。如果锁升级造成性能方面的问题，那么可能需要增大 locklist 参数或 maxlocks 参数的值。可以使用数据库系统监视器来确定是否发生了锁升级，

跟踪应用程序(连接)遭遇锁超时的次数或者数据库检测到的所有已连接应用程序的超时情形。

首先，运行下面的命令以打开针对锁的 DB2 监视器：

```
db2 -v update monitor switches using lock on
db2 -v terminate
```

然后收集数据库快照：

```
db2 -v get snapshot for database on DB_NAME
```

在快照输出中，检查下列各项：

```
Locks held currently = 0
Lock waits = 0
Time database waited on locks (ms) = 0
Lock list memory in use (Bytes) = 504
Deadlocks detected = 0
Lock escalations = 8
Exclusive lock escalations = 12
Agents currently waiting on locks = 0
Lock Timeouts = 0
Internal rollbacks due to deadlock = 0
```

如果“Lock list memory in use (Bytes)”超过定义的 locklist 大小的 50%，就增加 locklist 数据库配置参数中 4KB 页的数量。如果发生了“Lock escalations>0”或“Exclusive lock escalations>0”，就应该增加 locklist 或 maxlocks，抑或同时增加两者。查看“Locks held currently”、“Lock waits”、“Time database waited on locks (ms)”、“Agents currently waiting on locks”和“Deadlocks detected”中是否存在高值，如果有的话，就可能是差于最优访问计划、事务时间较长或应用程序并发问题的症状。为了发现死锁，需要创建针对死锁的事件监视器。事件监视器带有详细信息，以便查看当前正在发生的事情。锁升级、锁超时和死锁将表明系统或应用程序中存在某些潜在问题。锁定问题通常表明应用程序中存在一些相当严重的并发性问题，在增大锁列表参数值之前应当解决这些问题。指定应用程序在获得锁之前所等待的秒数(locktimeout)，这可以帮助避免全局死锁情况的发生。如果该值为－1，应用程序将会出现锁等待。对于生产系统中的 OLAP，一开始为 60 (秒)比较好；对于 OLTP，大约为 10 秒比较好。对于开发环境，应该使用－1 以识别和解决锁等待的情况。如果有大量的并发用户，那么可能需要增加 locktimeout 大小以避免频繁回滚。

如果“Lock Timeouts”是较高的数，那么可能由以下原因造成：

- locktimeout 的值太低
- 某个事务持有锁的时间有所延长
- 锁升级

下面是一些控制锁列表大小的建议：

- 经常进行提交以释放锁。
- 如果要执行大量更新，更新之前，在整个事务期间锁定整个表(使用 SQL LOCK TABLE 语句)。这里使用了一把锁，从而防止其他事务妨碍这些更新，但对于其他用户却减少了数据并发性。
- 使用 ALTER TABLE 语句的 locksize 参数控制如何在持久基础上对某个特定表进行锁定。
- 在业务逻辑允许的情况下确保应用程序使用最低的隔离级别。
- 查看应用程序使用的隔离级别。使用可重复读(Repeatable Read)隔离级别在某些情况下可能会导致自动执行表锁定。当有可能减少所持有共享锁的数量时，可以使用游标稳定性(Cursor Stability)隔离级别。如果没有损害应用程序完整性需求，那么可以使用未提交读(Uncommitted Read)隔离级别而不是游标稳定性隔离级别，从而进一步减少锁的数量。尽量使用 Cursor Stability 隔离级别(默认情况)，以便减少被持有的共享锁的数量(如果应用程序能够承受脏读，那么 Uncommitted Read 可以进一步减少锁)。

4.3.6 活动应用程序的最大数目(maxappls)

maxappls 是数据库配置参数，它指定了可以连接到数据库的并发应用程序(本地和远程)的最大数量。由于需要为连接到数据库的每个应用程序分配一些私有内存，因此允许有更多并发应用程序将意味着用掉更多内存。该参数的值必须大于等于，已连接应用程序的数量加上这些应用程序中在完成提交或回滚过程中可能并发存在的应用程序数量的总和。

当应用程序尝试连接数据库，但是连接到数据库的应用程序数已经达到了 maxappls 的值时，会向应用程序返回下面这个错误，表明连接到该数据库的应用程序数已达到了最大值：

```
SQL1040N The maximum number of applications is already connected to the
database. SQLSTATE=57030
```

建议使用默认值 automatic，即允许任何数目的已连接应用程序。数据库管理器将动态分配以支持新应用程序所需的资源。

4.3.7　pckcachesz

包缓存用作静态和动态 SQL 语句的缓存部分，它能够帮助数据库管理器减少内部开销，因为避免了在重新装载包时访问系统编目；或者，对于动态 SQL 来说避免了重新编译。

pckcachesz(用于定义包缓存大小的参数)应该大于“Package cache high water mark (Bytes)”。如果通过 db2 get snapshot for db on dbname 看到的“Package cache overflows”不为 0，那么可以尝试通过增加 pkgcachesz 来使这个计数器变为 0。

Package Cache Hit Ratio (PCHR)应该尽可能接近 100%(而不是从缓冲池中获取所需的内存)。可以用下面的公式来计算：

```
PCHR = (1-( "Package cache inserts"/ "Package cache lookups"))*100
```

4.3.8　catalogcache_sz

这个参数用于缓存系统编目信息，例如 systables、授权和 sysroutines 等信息。缓存编目信息十分重要，尤其是在使用 DPF 的情况下。因为不必为获得先前已经检索过的信息而访问系统编目(编目分区)，从而减少了内部开销。

不断增加该值，直到 OLTP 的 Catalog Cache Hit Ratio (CCHR)达到 95%或更好的值：

```
CCHR = (1-( "Catalog cache inserts"/ "Catalog cache lookups"))*100
```

如果“Catalog cache overflows”的值大于 0，也要增加该参数的值。还可以使用“Catalog cache high water mark(Bytes)”来确定编目缓存曾消耗过的最大内存。如果“High water mark”等于允许的 Maximum 大小，就需要增加编目缓存堆的大小。

4.3.9　异步页清除程序的数目(num_iocleaners)

num_iocleaners 是数据库配置参数，该参数指定数据库中异步页面清除器的数量。异步页面清除器用于将更改后的页面从缓冲池写到磁盘。一开始将这个参数设为等于系统中 CPU 的数量。当触发了 I/O Cleaners 时，它们会同时启动。因此，您可能不希望有那么多的清除器，以致影响性能和阻塞其他处理过程。num_iocleaners 还可以让您指定数据库中异步页清除程序的数目。在数据库代理程序需要缓冲池中的空间之前，这些页清除程序将缓冲池中已更改的页写到磁盘。这允许代理程序不必等待已更改页被写到磁盘就可以读取新页。因此，这会加快应用程序事务的执行。

如果将该参数设置成 0，将不启动页清除程序，结果是数据库代理程序将缓冲池中的所有页写到磁盘。该参数会对存储在多个物理存储设备上的单个数据库的性能产生显著影响，这是因为在这种情况下其中某个设备极有可能处于空闲状态。如果没有配置页清除程序，应用程序可能会遇到严重的性能问题。

如果连接到数据库的应用程序主要执行更新数据的事务，那么增加清除程序的数目会提高性能。增加页清除程序的数量还会减少“软”故障(比如断电)的恢复时间，因为磁盘上数据库的内容在任何给定时间都是比较新的。

当设置该参数的值时需要考虑下面这些因素：

- 如果有多个事务针对数据库运行，将该参数的值设置在 1 到该数据库所使用的物理存储器的数量之间。建议将该参数的值设置成至少您系统上 CPU 的数量。
- 在具有高更新事务率的环境下，可能需要配置较多的页清除程序。
- 在具有大缓冲池的环境下，也可能需要配置较多的页清除程序。

1. 如何更改该参数

可以用下面的命令来为该参数设置新值：

```
db2 -v update db cfg for sample using NUM_IOCLEANERS a_number
db2 -v terminate
```

2. 调优步骤

使用数据库系统监视器，利用有关从缓冲池进行写操作的快照数据(或事件监视器)信息来帮助您调优该配置参数。

当使用快照和收集缓冲池的快照数据时，监控下列计数器：

```
Buffer pool data writes = 12400
Asynchronous pool data page writes = 450
Buffer pool index writes = 239
Asynchronous pool index page writes = 201
LSN Gap cleaner triggers  = 0
Dirty page steal cleaner triggers  = 0
Dirty page threshold cleaner triggers = 0
```

如何决定该减少还是增加 num_iocleaners

如果 Asynchronous Write Percentage(AWP)是 90%或更高，减少 num_iocleaners；如果 Asynchronous Write Percentage(AWP)小于 90%，增加 num_iocleaners。

```
AWP = (( "Asynchronous pool data page writes"+ "Asynchronous pool index page
writes") * 100) / ( "Buffer pool data writes"+ "Buffer pool index writes")
```

“Dirty page steal cleaner triggers”指出因为在数据库“受损”缓冲区替换期间需要同步写操作而调用页清除程序的次数。为了有更好的响应时间，该数值应当尽可能低。利用

上面所示的计数器，可以使用下面的公式计算所有清除程序调用的百分比：

```
    Dirty page steal cleaner triggers / (Dirty page steal cleaner triggers +Dirty
page threshold cleaner triggers +LSN Gap cleaner triggers)
```

如果该比率很高，那么可能表明您所定义的页清除程序太少了。页清除程序太少会使故障恢复时间变长。

4.3.10　异步 I/O 服务器的数目(num_ioservers)

诸如备份和恢复之类的实用程序使用 I/O 服务器代表数据库代理程序执行预取 I/O 和异步 I/O 操作。该参数是数据库配置参数，用于指定数据库的 I/O 服务器的数目。超过这个数量的预取和实用程序 I/O 在任何时候都不能在数据库中执行。在启动 I/O 操作时，I/O 服务器处于等待状态。I/O 服务器用于执行预取操作，而此参数则指定数据库中 I/O 服务器的最多数量。非预取 I/O 是从数据库代理调度的，因此不受此参数的约束。

1. 如何更改该参数

可以使用下面的命令为 num_ioservers 设置新值：

```
db2 -v update db cfg for DB_NAME using NUM_IOSERVERS a_number
db2 -v terminate
```

2. 建议

一开始将该参数设置为数据库所跨的物理磁盘数(磁盘阵列中的许多磁盘或者逻辑卷中的许多磁盘)加上 1 或 2，但是不大于 CPU 数量的 4 到 6 倍。

如果很快就看到“Time waited for prefetch (ms)”，那么或许可以添加 IO 服务器，以查看性能是否有提高。

4.3.11　avg_appls

只有在应用程序发出复杂的 SQL(例如连接、函数、递归等)命令时才更改这个参数，否则让它一直为 1。这可以帮助您估计在运行时可以为访问计划提供多少缓冲池空间。该参数应该设为较低的值，即“Applications connected currently”的平均数乘以复杂 SQL 命令的百分比。当此参数设置为 automatic 时，在创建数据库配置文件或者重新设置数据库配置文件时，会立即将此参数更新为适当的值。设置此参数可以帮助优化器更准确地模拟缓冲池的使用量。如果您手动设置此参数，无论您运行的应用程序的平均数是多少，值都是从 2 开始。在您采用此设置来评估优化器的行为和测试应用程序的性能之后，可以按较小

的增量来增大此参数的值。每当您增大此参数的值时，请继续评估优化器的行为并测试应用程序的性能。将此参数的值设得太大可能会导致优化器低估可用于查询的缓冲池空间量。更改此参数的值之后，请考虑使用 REBIND PACKAGE 命令重新绑定应用程序。

4.3.12 chngpgs_thresh(DB)

使用这个参数可以指定缓冲池中被更改页面所占的百分比，此时将启动异步的页面清除器，将更改写入磁盘以便在缓冲池中为新的数据腾出空间。在只读环境下，不使用页面清除器。在 OLTP 中，使用 20 到 40 之间的值应该可以提高性能(在更新活动庞大的情况下使用 20)，因为使这个值更低一些将使 I/O Cleaners 从脏缓冲池页面写出数据次数更多，但是每次做的工作却变少了。如果没有很多的 INSERT 或 UPDATE，那么对于 OLTP 和 OLAP 来说，默认的 60 应该就比较好了。

如果“Dirty page steal cleaner triggers”是两位数，那么试着降低。如果“Buffer pool data writes”较高，而“Asynchronous pool data page writes”较低，那么试着降低这个参数。

DB2 还提供另一种页面清除算法，这种算法可以提高特定缓冲池的性能。您需要设置概要注册变量 db2_use_alternate_page_cleaning 为 on，这样将忽略 chngpgs_thresh。确保 num_ioservers 至少为 3，否则它会拖新算法的后腿。

4.3.13 maxfilop

这个参数用于指定每个数据库代理所能打开的最大文件数。如果打开文件时被打开的文件数超出了这个值，就要关闭该代理正在使用的一些文件。过度打开和关闭都会降低性能。SMS 表空间和 DMS 表空间文件容器都被视作文件来对待。通常 SMS 使用的文件要更多一些。

增加该参数的值，直到“Database files closed”为 0。

4.3.14 logprimary、logsecond 和 logfilsz

logprimary 指定要预先分配空间的主日志文件的数量，而 logsecond 则是按照需要来分配空间的。logfilsiz 用于定义每个日志文件的大小。

如果“Secondary logs allocated currently”的值很大，那么可能需要增加 logfilsiz 或 logprimary(但要确保 logprimary 加上 logsecond 的和不超过 256)。还可以使用“Maximum total log space used (Bytes)”来帮助指出对日志文件空间(主日志加上从日志)的依赖性。

日志文件的大小对灾难恢复有一定的影响，因为在灾难恢复中要使用日志发送(log shipping)。日志文件比较大时，性能会更好些，但是会潜在地加大丢失事务的几率。当主系统崩溃时，最近的日志文件及其事务可能无法发送到从系统，因为在失败之前没有关闭

该文件。日志文件越大，随着日志文件的丢失，丢失事务的可能性也越大。

为此参数选择的值取决于许多因素，包括正在使用的记录类型、日志文件的大小和处理环境的类型(例如，事务的长度和提交的频率)。增大此值将增大日志的磁盘要求，因为主日志文件就是在与数据库的第一个连接期间预分配的。如果您发现经常分配辅助日志文件，那么可通过增大日志文件大小(logfilsiz)或增大主日志文件的数目来提高系统性能。

结合 logfilsiz 设置修改日志数量，根据具体数据库需求，必须修改主日志的数量，比如较小的事务系统可以将 logprimary 设置为 13，对于较大的事务系统设置成 23 或更大。

对于定期需要大量日志空间的数据库，使用辅助日志文件。例如，每月运行一次的应用程序需要的日志空间可能会超过由主日志文件提供的日志空间。由于辅助日志文件不需要永久的文件空间，因此辅助日志文件在这种情况下有优势。

启用无限记录功能(将 logsecond 设置为 –1)之后，数据库管理器不会为需要回滚和写日志记录的事务保留活动日志空间。在回滚处理期间，如果活动日志路径和归档目标都已满(或者归档目标不可访问)，那么 blk_log_dsk_ful(“日志磁盘满时阻止进行日志记录”数据库配置参数)也应该设置为 enabled 以避免发生数据库故障。建议将 logsecond 初始值设置成 37 或更大的值。

4.3.15　stmtheap

语句堆用于在 SQL 语句编译期间作为编译器的工作区。对于每一条要处理的 SQL 语句，都要从该区域分配和释放空间。如果收到警告信息或错误信息，那么可以按 256 逐次增加该参数的值，直到错误消失。此参数对数据库性能影响重大，建议合理调整，如果您的应用程序收到 SQL0437W 警告，且查询的运行时性能不足，那么您可能要考虑增加 stmtheap 值(底层为 automatic 的值或固定值)，从而确保动态编程连接枚举能够成功。

4.3.16　dft_queryopt

用于指定在编译 SQL 查询时使用的默认优化级别。对于混合的 OLTP/OLAP，使用 5 或 3 作为默认值；对于 OLTP，使用更低的级别；而对于 OLAP，则使用更高的级别。对于简单的 SELECTS 或短的运行时查询(通常只需要花费不到 1 秒钟就可以完成)，使用 1 或 0 也许比较合适。如果有很多的表，有很多相同列上的连接谓词，那么请尝试级别 1 或 2。对于超过 30 秒钟才能完成的长时间运行的查询，或者如果要插入 UNION ALL VIEW，那么可以尝试使用级别 7。在大多数环境下都应该避免使用级别 9。

4.3.17　util_heap_sz (DB)

该参数指定 BACKUP、RESTORE 和 LOAD 实用程序可以同时使用的最大内存数。如

果正在使用 LOAD，那么对于每个 CPU，建议将 util_heap_sz 设置成至少 10000 页。为了提高备份、恢复和装载的速度，建议增大该参数。如果此参数设置得太低，您可能无法并行运行实用程序。应该根据需要动态更新此参数。如果实用程序较少，那么将此参数设置为较小的值。如果实用程序较多，或者实用程序消耗的内存较多，那么应该将此参数设置为较大的值。

4.4 调整 DB2 概要注册变量

DB2 概要注册变量通常会影响优化器和 DB2 引擎本身的行为。虽然概要注册变量有很多，但是其中的大部分都有非常特定的用途，因而在大部分的 DB2 环境中都不会用到。本节我们主要介绍的是一些影响性能的常用的概要注册变量。

表 4-2 列出了用于概要注册表一些基本管理命令：

表 4-2 概要注册表管理命令

命　令	描　述
db2set -all	列出当前设置的所有 DB2 注册变量
db2set -g | -i variable=value	设置指定的 DB2 注册变量，使其处于全局(-g)级或实例(-i)级

注意：

变量和值之间不要有空格，否则变量会重新被设置成默认值。

4.4.1 db2_parallel_io

默认值为 NULL。值：*TablespaceID*[:*n*]，……——定义表空间(由数字表空间标识识别)的逗号分隔列表。如果表空间的预取大小为 automatic，那么可以通过指定表空间标识，后面跟一个冒号，再加上每个容器中的磁盘数 *n*，向 DB2 数据库管理器表示该表空间的每个容器中的磁盘数。如果没有指定 *n*，那么使用默认值 6。

可以将 *TablespaceID* 替换为星号(*)来指定所有的表空间。例如，如果 db2_parallel_io=*，那么所有表空间都将使用 6 作为每个容器中的磁盘数。如果同时指定星号(*)和表空间标识，那么优先使用表空间标识设置。例如，如果 db2_parallel_io=*, 1:3，那么所有表空间都将使用 6 作为每个容器中的磁盘数，但是第一个表空间除外(它将使用 3 作为每个容器中的磁盘数)。

此注册变量用来更改 DB2 计算表空间的 I/O 并行性的方式。当启用了 I/O 并行性时(要

么通过使用多个容器来隐式启用，要么通过设置 db2_parallel_io 来显式启用)，通过发出正确数目的预取请求来实现此目标。每个预取请求都是对页的扩展数据块的请求。例如，表空间有两个容器，并且预取大小是扩展数据块大小的 4 倍。如果设置了此注册变量，那么对此表空间的预取请求将分为 4 个请求(每个请求针对一个扩展数据块)，并且可以使用 4 个预取程序并行处理这些请求。

如果表空间中的各个容器分布在多个物理磁盘上，或者表空间中的容器是在由多个物理磁盘组成的单个 RAID 设备上创建的，那么您可能需要设置此注册变量。

如果未设置此注册变量，那么任何表空间的并行度都是表空间的容器数。例如，如果将 db2_parallel_io 设置为 NULL，并且表空间有 4 个容器，那么将发出 4 个按扩展数据块大小计算的预取请求；或者，如果表空间有两个容器，并且预取大小是扩展数据块大小的 4 倍，那么对此表空间的预取请求将分为两个请求(每个请求对应两个扩展数据块)。

如果设置了此注册变量，并且表的预取大小不是 automatic，那么表空间的并行度为预取大小除以扩展数据块大小。例如，如果对预取大小为 160 而扩展数据块大小为 32 页的表空间设置了 db2_parallel_io，那么将发出 5 个按扩展数据块大小计算的预取请求。

如果设置了此注册变量，并且表空间的预取大小是 automatic，那么 DB2 使用下面的公式自动计算表空间的预取大小：

```
预取大小 =(容器数) * (每个容器的磁盘数) * 扩展数据块大小
```

表 4-3 总结了可用的不同选项和在每种情况下计算并行性的方式。

表 4-3 计算并行性的公式

表空间的预取大小	DB2_PARALLEL_IO 设置	并行性等同于
automatic(预取大小=容器数 * 1 * 扩展数据块大小)	未设置	容器数
automatic(预取大小=容器数 * 6 * 扩展数据块大小)	表空间标识	容器数 * 6
automatic(预取大小=容器数 * n * 扩展数据块大小)	表空间标识: n	容器数 * n
非 automatic	未设置	容器数
非 automatic	表空间标识	预取大小/扩展数据块大小
非 automatic	表空间标识：n	预取大小/扩展数据块大小

例如，假设您有 3 个表空间，标识分别为 3、4 和 5。它们的扩展数据块大小(extend size)为 4096 字节，每个表空间各有两个容器。表空间 3 和 4 的预取大小均为 automatic，而表空间 5 的预取大小为 16384 字节。假设您设置了 db2_parallel_io=*:5, 4:10，那么每个表空间的并行性为：

- 表空间 3　值 *n*(每个容器中的磁盘数)为 5，扩展数据块大小=4096，容器数=2，预取大小为 automatic。因此，预取大小为 2*5*4096 ，并行性=容器数* *n* =2*5=10。
- 表空间 4　注意此表空间的值 *n*(每个容器中的磁盘数)被特别设置为 10。扩展数据块大小=4096，容器数=2，预取大小为 automatic。因此，预取大小=2*10*4096，并行性=容器数**n*=2*10=20。
- 表空间 5　值 *n* 仍为 5，但它没有任何作用，因为预取大小不是 automatic。扩展数据块大小=4096，容器数=2，预取大小=16384。因此，并行性=预取大小/扩展数据块大小=16384/4096=4。

在某些情况下，使用此变量可能导致磁盘争用。例如，如果表空间有两个容器，每个容器有专用的单个磁盘，那么设置此注册变量可能导致这些磁盘发生争用，原因是两个预取程序将同时访问两个磁盘中的每个磁盘。但是，如果每个容器都分布在多个磁盘上，那么设置此注册变量将潜在允许同时访问 4 个不同的磁盘。

要激活对此注册变量的更改，请发出 db2stop 命令，然后输入 db2start 命令。

4.4.2　db2_evaluncommitted

该参数在“第 5 章：锁和并发”中有非常详细的描述，此处不在赘述。

4.4.3　db2_skipdeleted

该参数在“第 5 章：锁和并发”中有非常详细的描述，此处不在赘述。

4.4.4　db2_skipinserted

该参数在“第 5 章：锁和并发”中有非常详细的描述，此处不在赘述。

4.4.5　db2_use_page_container_tag

操作系统：所有操作系统。

默认值：NULL。

值：ON 或 NULL。

默认情况下，DB2 将容器标记存储在每个 DMS 容器的第一个扩展数据块中，而无论它是文件还是设备。容器标记是容器的元数据。在 DB2 V8.1 之前，容器标记存储在单

个页中。因此，它在容器中需要较少的空间。要继续将容器标记存储在单个页中，请将 db2_use_page_container_tag 设置为 ON。

但是，如果在对容器使用 RAID 设备时将此注册变量设置为 ON，那么将会降低 I/O 性能。因为对于您使用等于 RAID 条带大小或其倍数的扩展数据块大小创建表空间的 RAID 设备，将 db2_use_page_container_tag 设置为 ON 会导致扩展数据块不与 RAID 条带对齐。因此，I/O 请求可能需要访问相对于优化的物理磁盘更多的磁盘。强烈建议用户不要启用此注册变量，除非有非常严格的空间约束，或者您要求行为与 DB2 V8 之前的数据库行为保持一致。

要激活对此注册变量的更改，请发出 db2stop 命令，然后执行 db2start 命令。

4.4.6　db2_selectivity

操作系统：所有操作系统。

默认值：No。

值：Yes 或 No。

此注册变量控制在 SQL 语句的搜索条件中可以使用 SELECTIVITY 子句的情况。

当此注册变量设置为“Yes”时，可以为下列谓词指定 SELECTIVITY 子句：

- 至少有一个表达式包含主变量的基本谓词。
- 其中的 MATCH 表达式、谓词表达式或转义表达式包含主变量的 LIKE 谓词。

4.4.7　db2maxfscrsearch

操作系统：所有操作系统

- 默认值：5
- 值：-1 以及 1 到 33 554

指定在将记录添加至表中时，要搜索的可用空间控制记录(FSCR)的数量。默认情况是搜索 5 个 FSCR。修改此值允许您平衡插入速度与空间复用。使用较大的值将优化空间复用，使用较小的值将优化插入速度。将值设为-1 会强制数据库管理器搜索所有 FSCR。

4.5　内存自动调优

从 IBM DB2 V9 开始，一种新的内存调优特性(自调优内存管理)通过自动设置几个内

存配置参数值的方式，简化了内存配置任务。内存调整器一旦启用，就能够在几个内存使用者(包括排序、程序包缓存、锁定列表和缓冲池)之间动态地分配可用内存资源。

此调整器工作在 database_memory 配置参数定义的内存限制范围之内，而 database_memory 的值本身可在 Windows 和 AIX 平台上自动调优。当启用了针对 database_memory 的自调优之后(通过将其设置成 automatic)，DB2 内存调整器会确定数据库的总内存需求，并会根据当前的数据库需求增加或减少分配给数据库共享内存的容量。例如，如果当前的数据库需求很高，而系统又有足够的空闲内存，数据库内存就可以使用较多的内存资源。一旦数据库内存的需求降低，或系统的空闲内存量过低，一部分数据库共享内存就会被释放。

图 4-1 描述了在内存自动调优机制中数据库与操作系统之间内存互动的基本流程。

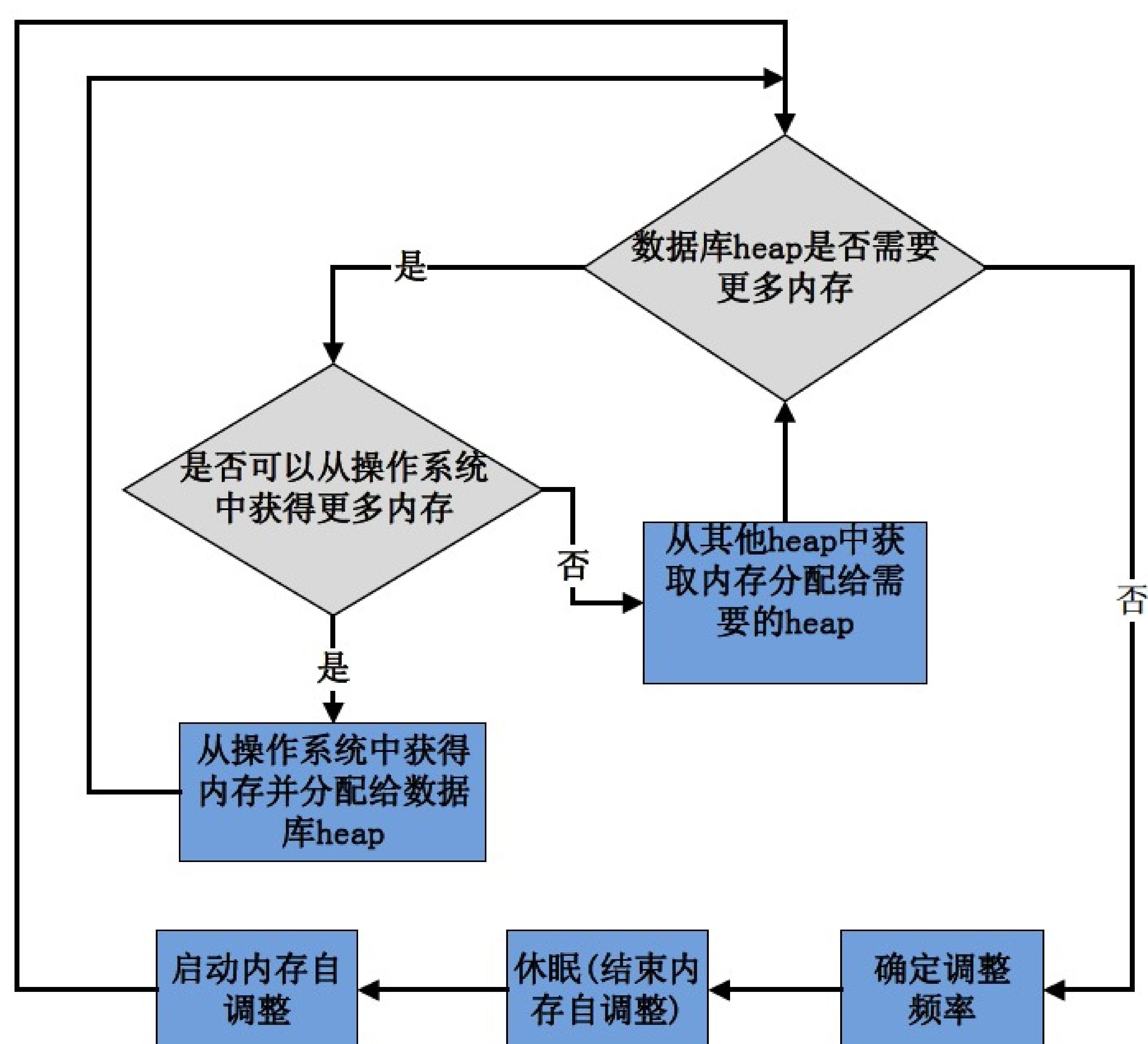


图 4-1　内存自动调优机制

4.5.1　内存自动调优示例

图 4-2 显示的示例来自启动了内存自动调优的数据库，此例来源于 IBM。

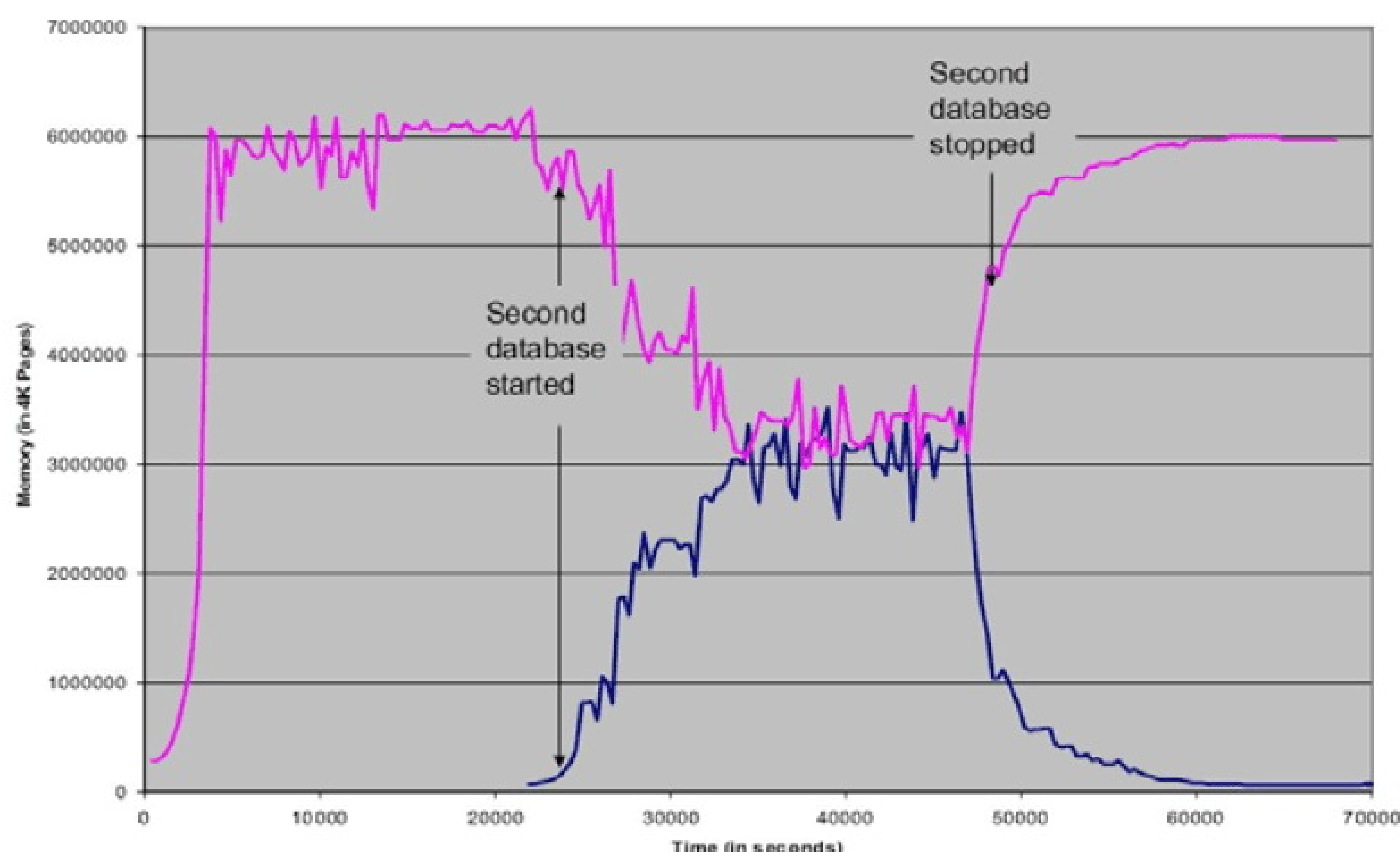


图 4-2　内存自动调优的效果

在上述示例中，第一个数据库启动的时候，数据库分配的共享内存很少，但是随着数据库负载的增加，内存的需求也在不断增加。由于在操作系统级别没有其他的应用参与内存的竞争，因而可以不断地从操作系统中获得内存，直到不能再获得为止。

当第二个数据库启动的时候，数据库分配的共享内存同样很少。同样，随着数据库负载的增加，内存的需求也在不断增加。但是操作系统中已经没有空闲的内存能够分配了，所以只能从 DB2 的其他 heap 中获取内存，第一个数据库中已经分配的内存会被逐渐转移到第二个数据库中，直到两个数据库达到一定程度的平衡。

在第二个数据库停止之后，由于不再有内存的竞争者，因此第一个数据库又会逐步地获得所有的内存，从而获得最大的数据处理能力。

上面的示例充分揭示了 DB2 的内存自动调优机制如何在数据库与操作系统之间、数据库与数据库之间自动地调整内存的分配，以使数据库在一般意义上获得足够、合理的内存资源。

4.5.2　启用内存自动调优及相关参数

1. 数据库内存配置

DB2 中的参数 database_memory 用于控制数据库中内存总量的值，通过将 database_memory 设置成不同的值可以实现数据库级别的内存自动调优。具体设置方式有如下几种：

- 将 database_memory 设置为 automatic，此时，DB2 中共享内存的大小可以与操作系统内存之间自动地进行调整。也就是说，当操作系统中有空闲内存的时候，数据库可以从操作系统中获得内存以满足负载的需求；反之，当数据库中有不需要的内存的时候，就可以将其返还给操作系统。
- 将 database_memory 设置为 computed，此时，DB2 的内存总量为数据库中设置的所有内存之和再增加 20%。
- 将 database_memory 设置为大于 1.2 倍实际内存总和的数值，此时，如果数据库中某个设置为 automatic 的内存 heap 需要更大的内存，就可以使用大于总量 1.2 倍的那部分内存。

在将 database_memory 设置为 automatic 之后，数据库中所有设置为 automatic 的内存 heap 就可以获得最大的灵活性。

在数据库中可以设置为自动调优的主要的内存 heap 如下：

- sortheap
- pckcachesz
- locklist

2. 缓冲池内存配置

缓冲池是用来在内存中缓存数据页的数据库对象。一旦数据页被放入缓冲池，就可以避免对硬盘的物理 I/O 访问。缓冲池可以配置成只缓存某一表空间。缓冲池是 DB2 性能调优中非常重要的部分。如何配置系统的实际和虚拟内存、DB2 的内部内存使用，以及 DB2 缓冲池会极大影响系统的性能。可用实际内存越多，整体的性能越好。您可针对自己系统处理器的数量对这些因素进行均衡。实际内存不足会导致操作系统产生过多的分页，而这又会影响到每个应用程序，包括 DB2。即使是一页 DB2 数据已经在缓冲池内，这种情况也仍会发生，原因是操作系统没有足够的内存来在实际内存中保存该页，操作系统必须将它暂时写出到硬盘中。这种情况会严重影响 DB2 的性能。

3. 实例内存配置参数

DB2 中实例内存配置参数 instance_memory 指定可以为数据库分区分配的最大内存量。默认值 automatic 将允许实例内存根据需要而增加，在激活数据库分区时，自动计算的值介于物理 RAM 的 75%到 95%之间。在多分区数据库的服务器上，该计算值是除以逻辑分区数而获得的值。

当 instance_memory 设置为特定值时，数据库管理器分配的系统内存最多只能为此限制。当达到此限制时，应用程序会接受到内存分配错误。如果启用了自调整 database_memory，那么自调整管理器(STMM)将更新配置以获得最佳性能，同时还会监视可用系统内存。通过监视可用内存，可确保不会过量使用系统内存。这里需要注意的是，审计缓冲区(audit_buf_sz)的大小不受 instance_memory 的限制。

当 instance_memory 设置为 automatic 时，数据库管理器将根据需要来分配系统内存。如果启用了自调整 database_memory，那么 STMM 将更新配置以获得最佳性能，同时还会监视可用系统内存。通过监视可用内存，可确保不会过量使用系统内存。

通过使用 db2pd -dbptnmem 命令或 admin_get_mem_usage 表函数，我们可以非常方便地查看实例内存耗用总量以及当前 instance_memory 耗用量。

4.5.3　内存配置参数的配置原则

自 DB2 V9 以后，大部分内存参数的默认值都是 automatic，这大量节约了数据库维护人员的精力。当启用自调整内存功能并对大多数的参数使用 automatic 设置时，我们需要理解清楚每个配置参数的含义以及相互的关联性，这样我们就可以更好地控制它们的设置，能够在特定场景中找到“内存不足”的原因。

基本上，DB2 数据库管理器使用两种类型的内存：

- **性能内存**：这是用来提高数据库性能的内存。性能内存由自调整内存管理器(STMM)控制并分发给各种性能堆。可以将 database_memory 配置参数设置为性能内存的最大容量，也可以将 database_memory 设置为 automatic 以便 STMM 管理性能内存的全部容量。
- **功能内存**：此内存由应用程序使用。可以使用 appl_memory 配置参数来控制 DB2 数据库代理程序为了为应用程序请求提供服务而分配的功能内存(应用程序内存)的最大容量。默认情况下，此参数设置为 automatic，这意味着只要有系统资源可用，就允许功能内存请求。如果要使用具有内存使用限制的 DB2 数据库产品，或者将 instance_memory 设置为特定值，那么在数据库分区分配的内存总量未超过 instance_memory 限制的情况下，将强制使用 instance_memory 限制并且允许功能内存请求。

实例和数据库内存自调整的关联

在 automatic 设置可用之前，可以使用各种操作系统或 DB2 工具(db2mtrk)来查看不同类型的内存(例如共享内存、专用内存、缓冲池内存、锁定列表、排序内存(堆)等等)耗用的

空间量。如果其中某个堆达到内存限制，那么应用程序中的某条语句将失败，并且显示“内存不足”错误消息。现在，可以使用缺省的 automatic 配置参数来去除各个功能内存堆的硬上限。

当自调整内存管理器(STMM)处于活动状态并且启用了数据库内存的自调整功能时，STMM 将检查系统中的可用内存量，并自动确定为了获取最佳性能而应该供性能堆专用的内存量。所有性能堆都将计入 database_memory 总大小。除了性能内存需求以外，还需要一定内存来确保 DB2 数据库管理器的操作和完整性。如果未施加 instance_memory 限制，那么对单个应用程序可分配的内存量没有其他限制。

如果存在 instance_memory 限制，STMM 还会定期查询剩余的可用系统内存量以及剩余的可用 instance_memory 空间量。为了防止应用程序发生故障，STMM 认为应用程序需求优先于性能条件。有需要时，将通过减少可供性能堆使用的空间量使性能下降，从而提供足够的可用系统内存和 instance_memory 空间以满足应用程序内存请求。应用程序完成时，使用的内存将被释放，从而可供其他应用程序重复使用或者为 database_memory 回收以供 STMM 使用。

根据以上对 instance_memory 和 database_memory 的描述，建议在生产数据库环境中，把 instance_memory 设置成固定值。通过此硬上限，可以避免数据库和应用程序过多抢占主机的内存资源。在数据库级别则建议把 database_memory 设置成 automatic，让自调整内存管理器调节不同类型的内存。

4.6 本章小结

以上列出了在数据库管理过程中，与性能有关的最主要的配置参数，这些参数分为实例参数、DB 参数和 DB2 注册变量。其中，DB2 注册变量的有效范围可以是全局级别或实例级别；实例参数的有效范围是整个实例，这包括了实例中创建的所有数据库；DB 参数的有效范围是当前的数据库。

在实际的生产环境中，正确、合理地设置这些参数对数据库的性能影响至关重要，不合理的参数设置往往会造成严重的性能问题。

在性能问题上，数据库参数的调整往往不是简单、一次性的步骤，而应该是循序渐进、循环处理的过程。因为在发生性能问题需要对参数进行调整的时候，很少能确定地知道只要将某个或几个参数设置为特定的值就可以达到最好的效果。为了达到最好的调整效果，就需要由调整到监控、再到调整的持续循环过程。所以性能参数的调整通常具有持续性的特点。

第 5 章

锁 和 并 发

我们在进行性能问题分析时遇到最多的问题之一就是锁。“为什么 DB2 锁住了这个表、行或对象？”，“这个锁会阻塞多长时间及为什么？”，“为什么出现了死锁？”，“我的锁请求在等待什么？”，诸如此类等。仔细地分析一些常见的锁示例，可以说明 DB2 锁定策略背后的原则。很多 DB2 用户都会碰到有关锁等待、死锁和锁升级等与锁相关的问题。在本章中，我们将会对这些问题进行详细讲解，并介绍如何解决这些问题。

本章主要讲解以下内容：

- 锁的概念
- 锁的属性、策略和模式
- 隔离级别的概念及使用
- 锁转换、锁等待、锁升级和死锁
- 与锁有关的性能问题
- 锁与应用程序设计
- 锁监控工具及应用案例
- 最大化并发性

5.1　锁的概念

5.1.1　数据一致性

理解数据一致性

什么是数据一致性？我们通过示例来回答这个问题。假定你的公司拥有多家连锁饭店，公司使用数据库来跟踪每家饭店中的货物存储量。为了使货物的采购过程更方便，数据库中包含了每个连锁店的库存表。每当一家饭店收到或用掉一部分货物时，与该饭店相对应的库存表就会被修改以反映库存变化。

现在，假定从一家店调配若干瓶番茄酱到另一家店。为了准确地表示这一次库存调配，调出方饭店表中存储的番茄酱瓶数必须减少，而接收方饭店表中存储的番茄酱瓶数必须增加。如果用户减少调出方饭店库存表中的番茄酱瓶数，但没有增加接收方库存表中的番茄酱瓶数，数据就会出现不一致。此时，所有连锁店的番茄酱的总瓶数就不准确了。

如果用户忘记了进行所有必要的更改(正如前面示例中的一样)，或者在进行更改的过程中系统崩溃了，又或者数据库应用程序由于某种原因过早地停止了，那么数据库中的数据就都变得不一致。当几个用户同时访问相同的数据库表时，也可能发生不一致。为了防止数据的不一致(尤其是在多用户环境中)，DB2 的设计中集成了下列数据一致性支持机制：

- 事务
- 锁
- 隔离级别

5.1.2　事务和事务边界

事务(也称为*工作单元*)是一种将一个或多个 SQL 操作组合成单元的可恢复操作序列，通常位于应用程序进程中。事务的启动和终止定义了数据库的一致性：要么将事务中执行的所有 SQL 操作的结果都应用于数据库(提交)，要么完全取消并丢弃已执行的所有 SQL 操作的结果(回滚)。

运行嵌入式 SQL 应用程序或脚本，在可执行 SQL 语句第一次执行时(在建立与数据库的连接之后或在现有事务终止之后)，事务就会自动启动。事务在启动之后，必须由启动事务的用户或应用程序显式地终止，除非使用了称为自动提交(automatic commit)的过程(在这种情况下，发出的每条单独的 SQL 语句被看作单个事务，一执行就被隐式地提交了)。

在大多数情况下，通过执行 COMMIT 或 ROLLBACK 语句来终止事务。当执行

COMMIT 语句时，自从事务启动以来对数据库所做的一切更改就成为永久性的了——它们被写到磁盘。当执行 ROLLBACK 语句时，自从事务启动以来对数据库所做的一切更改都被撤销，并且数据库返回到事务开始之前所处的状态。不管是哪种情况，数据库在事务完成时都保证能回到一致状态。

1. COMMIT 和 ROLLBACK 操作的效果

正如前面提到的，通常通过执行 COMMIT 或 ROLLBACK SQL 语句来终止事务。为了理解这些语句如何工作，我们看下面的示例。

如果按所示的顺序执行下列 SQL 语句(由 3 个事务组成的简单工作负载)：

```
CONNECT TO SAMPLE
CREATE TABLE DEPARTMENT (DEPT ID INTEGER NOT NULL, DEPT NAME VARCHAR(20))
INSERT INTO DEPARTMENT VALUES(100, 'PAYROLL')
INSERT INTO DEPARTMENT VALUES(200, 'ACCOUNTING')
COMMIT

INSERT INTO DEPARTMENT VALUES(300, 'SALES')
ROLLBACK

INSERT INTO DEPARTMENT VALUES(500, 'MARKETING')
COMMIT
```

这将创建名为 DEPARTMENT 的表，结构如表 5-1 所示。

表 5-1　表 DEPARTMENT 的结构

DEPT_ID	DEPT_NAME
100	PAYROLL
200	ACCOUNTING
500	MARKETING

当执行第 1 条 COMMIT 语句时，创建名为 DEPARTMENT 的表并向表中插入两条记录，这两个操作都会变成永久性的。当执行到 ROLLBACK 语句时，删除插入 DEPARTMENT 表中的第 3 条记录，该表返回到执行插入操作之前所处的状态。最后，当执行到第 2 条 COMMIT 语句时，插入到 DEPARTMENT 中的第 4 条记录成为永久性的，而数据库再次返回到一致状态。

从上面这个示例中可以看出，提交或回滚操作只影响在这个操作所结束的事务内做出

的更改。只要数据更改仍然未被提交，其他用户和应用程序通常就无法看见它们(也有例外情况，在本章的最后我们将进行讨论)，并可以通过执行回滚操作取消它们。但是，一旦数据更改被提交，其他用户和应用程序就可以访问它们，并且再也不能通过回滚操作取消它们。

2. 不成功事务的效果

我们刚才已看到当通过 COMMIT 或 ROLLBACK 语句终止事务时会发生什么。但是，如果在事务完成前出现系统故障，那会发生什么情况呢？在这种情况下，DB2 数据库管理程序会取消所有未提交的更改，从而恢复数据库一致性(假定在事务启动时就存在这样的一致性)。图 5-1 对比了成功的事务和在成功终止之前失败的事务的效果。

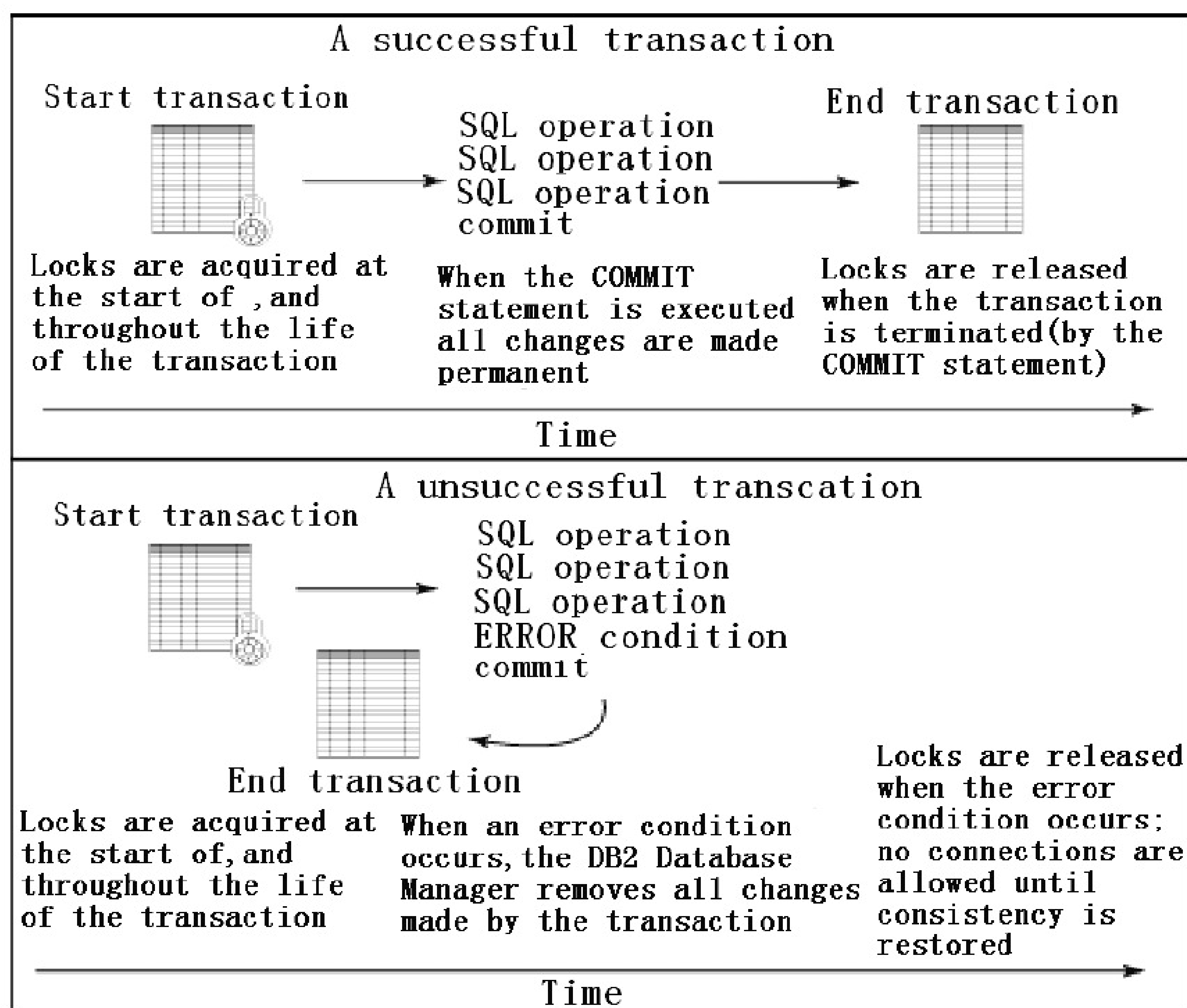


图 5-1　对比成功的和不成功的事务

5.1.3　锁的概念

在关系数据库(DB2、Oracle、Sybase、Informix 和 SQL Server 等)中，最小的恢复和交易单位为事务，事务具有 ACID(原子性、一致性、隔离性和永久性)特征。关系数据库为了

确保并发用户在存取同一数据库对象时的正确性(无丢失更新、可重复读、不读“脏”数据、无“幻像”读)、数据库中引入了并发(锁)机制。

成熟的关系数据库系统应该能够允许多个应用程序同时对相同数据进行访问。当这种情况发生时，要保证数据库的完整性，就必须有一定的机制用于控制数据记录的读取、插入、删除和更新。锁就是这样一种机制。基本的锁有两种类型：排它锁(exclusive locks，记为 X 锁)和共享锁(share locks，记为 S 锁)。

锁是一种用来将数据资源与单个事务关联起来的机制，其用途是当某个资源与拥有它的事务关联在一起时，控制其他事务如何与资源进行交互。我们称与被锁定资源关联的事务持有或拥有该锁，DB2 数据库管理程序用锁来禁止事务访问其他事务写入的未提交数据(除非使用了未提交的读隔离级别)，并禁止其他事务在拥有锁的事务使用限制性隔离级别时对这些行进行更新。一旦获取锁，在事务终止之前，就一直持有锁；在事务终止时释放锁，这样其他事务就可以使用被解锁的数据资源了。

如果一个事务尝试访问数据资源的方式与另一个事务持有的锁不兼容(稍后我们将研究锁兼容性)，那么这个事务必须等待，直到拥有锁的事务终止为止。这被称为锁等待。当锁等待事件发生时，尝试访问数据资源的事务所做的只是停止执行，直到拥有锁的事务终止或不兼容的锁被释放为止。

举个例子，假如事务 T 对数据 D 加 X 锁，则其他任何事务都不能再对 D 加任何类型的锁，直至 T 释放 D 上的 X 锁；一般要求在修改数据前要向数据加排它锁，所以排它锁又称为写锁。若事务 T 对数据 D 加 S 锁，则其他事务只能对 D 加 S 锁，而不能加 X 锁，直至 T 释放 D 上的 S 锁；一般要求在读取数据前要向数据加共享锁，所以共享锁又称为读锁。我们可以通过调整数据库的加锁策略来适应一定的并发性需求。

通过对数据库对象加锁，我们可以避免以下情况的发生：

由于并发更改造成数据的丢失

假设航空机票代理 1 查询 512 航班的所有空闲座位，发现有一个座位空着，这时航空机票代理 1 为某位乘客预定该座位；与此同时，航空机票代理 2 也发现了这个空的座位，并且也为其他乘客预定，结果这两个航空机票代理都收到预定成功的结果，但是事实上，后提交的预定会将先提交的预定修改掉。

让我们用数据库来对上述情况再进行一下细致描述：

P Read 和 Instuct 两人同时来到不同的航空代理处购买 512 航班的机票，两个代理都想为自己的顾客预订 7A 座位，然后两个代理同时输入下列命令：

代理 1：

```
    update reservations set p name='P Read'where flight='512'and seat='7A' and
p_name is null;:
```

代理 2：

```
    update reservations set p name='Instuct' where flight='512' and seat='7A'
and  p_name is null;:
```

如果没有一定的机制来阻止对同一数据的并发更改，两个代理都会收到更改成功的信息。P Read 和 Instuct 将会在机场出现并认为他们都已经预订好座位。

如果使用加锁机制进行控制，这种情况可以避免。当 P READ 的代理在更新座位信息的时候，在表中先使用排它锁对数据进行锁定，然后更新，更新结束后再将锁释放，那么 Instuct 的代理就不可能获得同样的座位信息。

读取了未提交的数据(脏读)

当可以读取未提交的数据时，可能会出现“脏读”，因为如果这些数据的修改被回滚，就会导致在数据处理中使用无效数据。

图 5-2 说明了这种场景。在这里，更新了一行并更改了 P-Name 字段，但是没有提交。然后，另一个使用 UR 隔离级别的应用程序执行 SELECT 语句。因为 UR 忽略行上的锁，所以它将返回未提交的值。但是，更新事务在提交之前被回滚。因而，SELECT 语句返回错误的值。

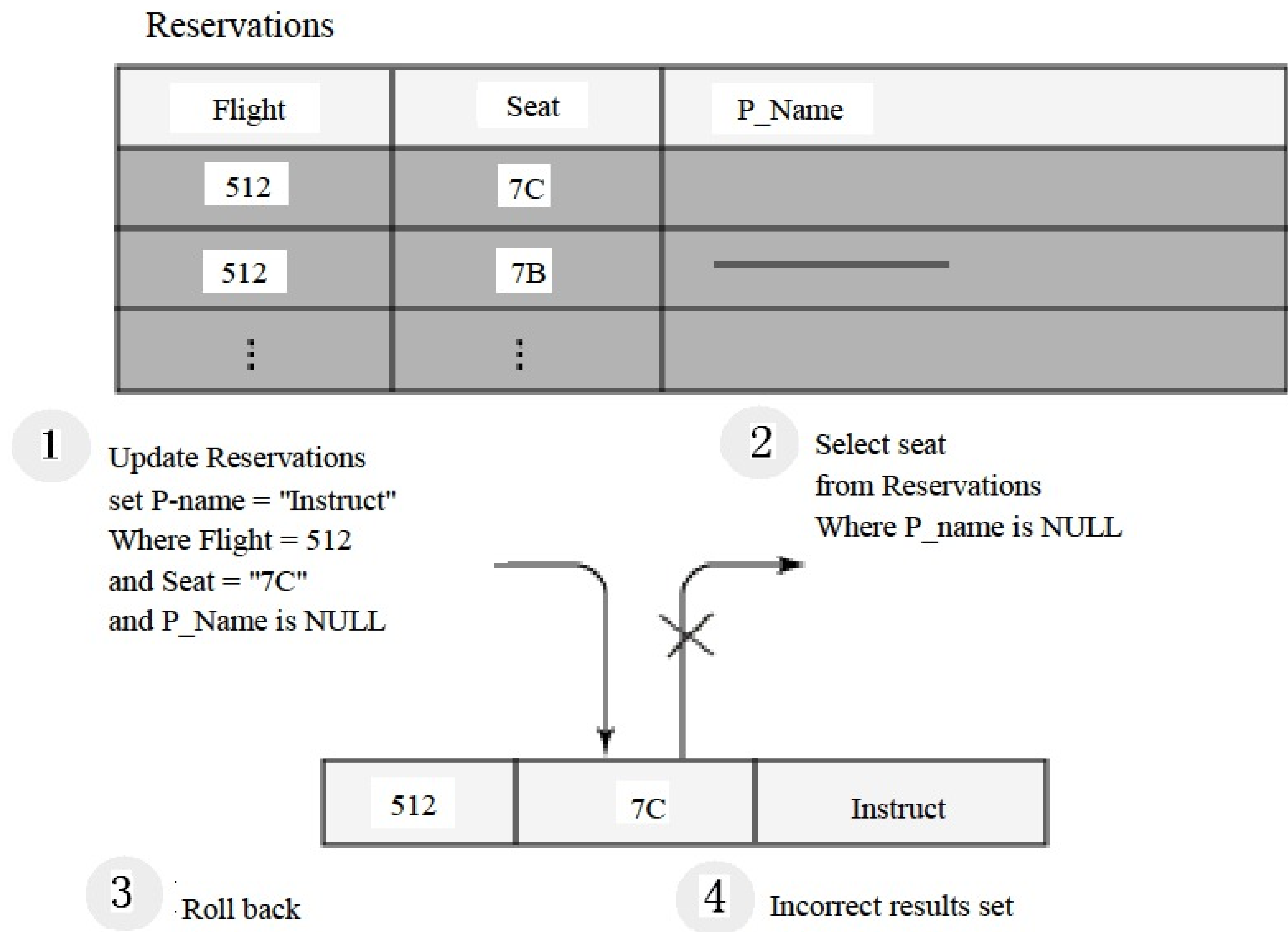


图 5-2　读取了未提交的数据

未提交的数据是指被命令更改、但还未被提交的数据。在很多情况下，如果允许用户对未提交的数据进行存取而数据最终却未被提交，那么用户据此做出的决定有可能是错

误的。

想象如下场景：在 512 航班上 7C 是唯一剩下的座位。现在 Instruct 来到某个航空代理处想购买 512 航班的机票，该代理发现 7C 是唯一剩下的座位，于是键入下列语句：

```
    update reservations set p name='Instruct' where flight=512 and seat='7C'
and p_name=NULL;:
```

这时，P Read 来到另外一个航空代理处想购买 512 航班的机票，该代理输入下列查询命令：

```
    select seat from reservations where p_name is NULL;:
```

如果不阻止对未提交数据的读取，那么由于 7C 已经被 Instruct 预订了，所以查询语句返回的记录数为零，意味着该航班已经客满，P READ 就不能买到座位。

过了一会儿，Instruct 认为该航班太贵，于是，该事务相应地回滚，7C 座位被空出。如果数据库通过加锁机制阻止对未提交数据的存取，P Read 就可以买到 7C 座位。

DB2 可以通过加锁机制来阻止对未提交数据的访问。当然，在某些情况下，对未提交数据的访问应该是允许的。

防止不可重复读

如果相同查询在同一工作单元中返回不同的结果集，就是出现了“不可重复读”。图 5-4 中说明了这种场景。假设航空机票代理 1 查找从 Dallas 到 Honolulu 的所有航班，航空机票代理 2 在 flight 表中删除了一个航班，因为这个航班被取消了。如果航空机票代理 1 在它的事务中再次执行同样的查询，就不会得到完全相同的数据集：其中不包含取消的那个航班。因为在航空机票代理 1 运行其事务时，允许航空机票代理 2 更改这个表，这个查询将是不可重复读的。

FLIGHT	SEAT	NAME	DESTINATION	ORIGIN
512	7C		DENVER	DALLAS
⋮		——		
⋮				
814	8A	——	SAN JOSE	DENVER
⋮				
134	1C	——	HONOLULU	SAN JOSE
⋮				⋮

Book a flight from Dallas to Honolulu

图 5-3　不可重复读

不可重复读是指由于在同一事务中，两次执行同样的 SELECT 语句，得到的结果却不同。这种情况会导致原先做出的决定由于条件的更改而产生偏差。

例如表 5-2 所示的航班表：

表 5-2 航 班 表

航 班	座 位	旅客名称	目的地	出发 地
512	7C		DENVER	DALLAS
814	8A		SAN JOSE	DENVER
134	1C		HONOLULU	SAN JOSE

想象以下场景：Instruct 需要从 DALLAS 飞往 HONOLULU，他找到航空代理处 1，该代理发现没有直达航班，但可以从 DALLAS 到 DENVER，再到 SAN JOSE，最后到达 HONOLULU(如表 5-2 所示)。Instruct 正在决定该路线是否可行时，P Read 从另外一个航空代理处 2 预订了 814 航班的 8A 座位，而这恰好是该航班的最后一个座位。如果这时 Instruct 发现代理提供的路线可行，决定购买时，却发现该路线已经不能成立，Instruct 就必须重新选择路线。

采用加锁机制，DB2 能够支持可重复读请求，可以允许或阻止应用程序修改其他应用程序正在访问的数据。对于上述场景而言，就是当 Instruct 在做决定时，禁止航空代理处 2 访问和更改 814 航班的 8A 座位信息，因为代理处 1 已经加了锁。

幻像读

如果应用程序在事务中多次执行同一 SQL 语句，而且后续执行会返回附加行，就是发生了“幻像读”。

图 5-4 中说明了这种场景。假设航空机票代理 1 查询 512 航班的所有空闲座位，发现只有一个座位空着。在此之后，由于一位乘客取消了飞机票，航空机票代理 2 取消了这个航班上某个座位的预订。如果代理 1 在这个事务中再次执行同一查询，那么不会得到与上次查询完全相同的数据集——因为出现了新的空闲座位。

让我们用数据库来对上述情况再进行一下细致描述：Instruct 来到某航空代理处购买 512 航班的机票，发出查询命令后返回一个空的座位。Instruct 正在考虑是否需要预定的时候，P Read 取消了之前预定的 512 号航班。当 Instruct 重新执行查询操作时，发现又多出了一个空闲座位。

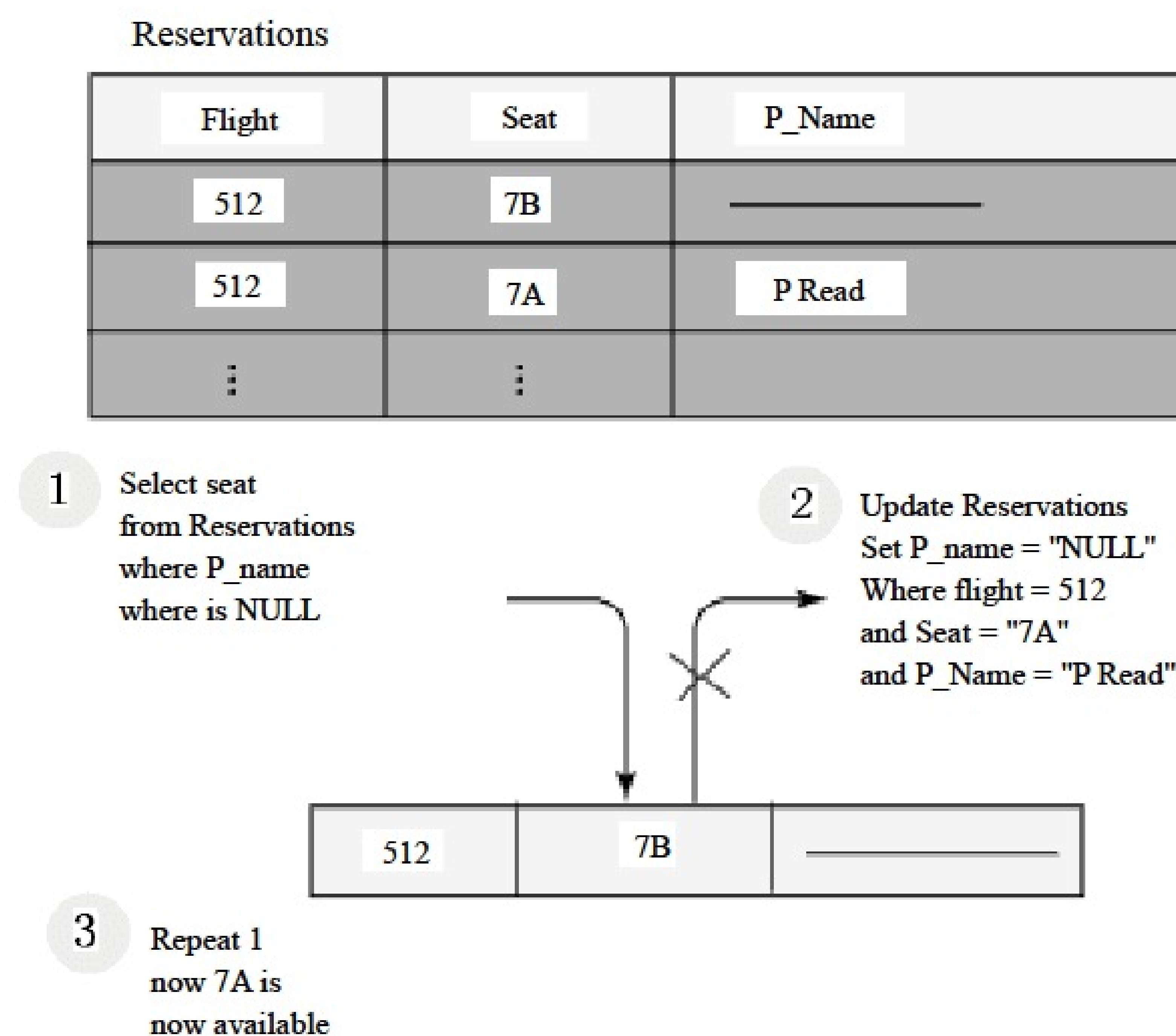


图 5-4　幻像读

5.2　锁的属性、策略及模式

5.2.1　锁的属性

所有的锁都具有下列基本属性：

- **对象**：该属性标识要锁定的数据资源。DB2 数据库管理程序在需要时锁定数据资源(如表空间、表和行)。DB2 支持对表空间、表、行和索引加锁(大型机上的 DB2 还支持对数据页加锁)来保证数据库的并发完整性。不过，在考虑用户应用程序的并发性这一问题上，通常并不检查用于表空间和索引的锁。分析此类问题的焦点在于表锁和行锁。
- **锁定数**：该属性指定持有锁的时间长度。事务的隔离级别通常控制着锁的持续时间。
- **方式**：该属性指定允许锁的拥有者执行的访问类型，以及允许并发用户对被锁定数据资源执行的访问类型。这个属性通常称为*锁状态*。

5.2.2　加锁策略

DB2 可以只对表进行加锁，也可以对表和表中的行进行加锁。如果只对表进行加锁，那么表中所有的行都会受到同等程度的影响。如果加锁的范围针对表及下属的行，那么在

对表加锁后，相应的数据行上还要加锁。应用程序究竟是对表加行锁还是同时加表锁和行锁，是由应用程序执行的命令和系统的隔离级别确定的。如果某个应用程序挂起某个数据库对象上的锁定，那么另一个应用程序就可能无法访问该对象。因此，锁定最少量数据并使这些数据不可访问的行级别锁定相比表级别而言，对于最大化并行性更好。但是，锁定需要内存和处理时间，因此单个表锁定可以最小化锁定开销。

5.2.3 锁的模式

DB2 在表一级加锁可以使用表 5-3 所示的方式。

表 5-3 表一级的加锁方式

名称缩写	全 名	描 述
IN	无意图(Intent None)锁 不需要行锁	该锁的拥有者可以读表中的任何数据，包括其他事务尚未提交的数据，但不能对表中的数据进行更改
IS	意图共享(Intent Share)锁 需要行锁配合	该锁的拥有者在拥有相应行上的 S 锁时可以读取该行的数据，但不能对表中的数据进行更改
IX	意图排它(Intent eXclusive)锁 需要行锁配合	该锁的拥有者在拥有相应行的 X 锁时可以更改该行的数据
SIX	共享携意图排它 (Share with Intent eXclusive)锁 需要行锁配合	该锁的拥有者可以读表中的任何数据，如果在相应的行上能够获得 X 锁，那么可以修改该行。SIX 锁的获得比较特殊，是在应用程序已经拥有 IX 锁的情况下请求 S 锁，或是在应用程序已经拥有 S 锁的情况下请求 IX 锁时生成的
S	共享(Share)锁 不需要行锁配合	该锁的拥有者可以读表中的任何数据，如果表被加上 S 锁，该表中的数据就只能被读取，不能被改变
U	更新(Update)锁 不需要行锁配合	该锁的拥有者可以读表中的任何数据，在升级到 X 锁之后，还可以更改表中的任何数据。该锁是处于等待对数据进行更改的一种中间状态
X	排它(eXclusive)锁 不需要行锁配合	该锁的拥有者可以读取或更改表中的任何数据。如果对表加上 X 锁，除了未提交读程序外，其他应用程序都不能对该表进行存取
Z	超级排它 (Super Exclusive)锁 不需要行锁配合	该锁不是通过应用程序中的 DML 语言生成的。一般是通过对表进行删除(Drop)和转换(Alter)操作或创建和删除索引而获得的。如果对表加上 Z 锁，其他应用程序(包括未提交读程序)就不能对该表进行存取

下面对表5-3中几种表锁的模式作进一步阐述：

IS、IX、SIX方式用于表一级并需要行锁配合，它们可以阻止其他应用程序对该表加上排它锁。

- 如果一个应用程序获得某表的IS锁，该应用程序可获得某一行上的S锁，用于只读操作，同时其他应用程序也可以读取该行，或是对表中的其他行进行更改。
- 如果一个应用程序获得某表的IX锁，该应用程序可获得某一行上的X锁，用于更改操作，同时其他应用程序可以读取或更改表中的其他行。
- 如果一个应用程序获得某表的SIX锁，该应用程序可以获得某一行上的X锁，用于更改操作，同时其他应用程序只能对表中的其他行进行只读操作。

S、U、X和Z方式用于表一级，但并不需要行锁配合，是比较严格的表加锁策略。

- 如果一个应用程序得到某表的S锁，该应用程序可以读表中的任何数据，同时允许其他应用程序获得该表上的只读请求锁。如果有应用程序需要更改或读该表中的数据，就必须等S锁被释放。
- 如果一个应用程序得到某表的U锁，该应用程序可以读表中的任何数据，并最终可以通过获得表上的X锁来得到对表中任何数据的修改权。其他应用程序则只能读取该表中的数据。U锁与S锁的区别主要在于更改意图上。U锁的设计主要是为了避免两个应用程序在拥有S锁的情况下同时申请X锁而造成死锁。
- 如果一个应用程序得到某表上的X锁，该应用程序可以读或修改表中的任何数据。其他应用程序不能对该表进行读或更改操作。
- 如果一个应用程序得到某表上的Z锁，该应用程序可以读或修改表中的任何数据。其他应用程序，包括未提交读程序都不能对该表进行读或更改操作。

注意：

对于IN、IX、IS和SIX这些意图锁，读者可以这样理解：严格来说它们并不是一种锁，而是存放表中行锁的信息。举个通俗的例子，我们去住酒店。可以把整个酒店比喻成一张表，每个房间是一行。那么当我们预订房间时，就对该行(房间)加X锁，但是同时会在酒店的前台对该行(房间)做信息登记(旅客姓名、住多长时间等)。大家可以把意图锁当成这个酒店的前台，它并不是真正意义上的锁，它维护表中每行的加锁信息，是共用的。后续的旅客通过酒店前台来看哪个房间是可住的，那么，如果没有意图锁，会出现什么情况呢？假设我要预订房间，那么每次我都需要到每一个房间查看以确认这个房间有没有住旅客，显然这样做的效率是很低下的。其实，最早的DB2版本是没有意图锁的，但是这对并发影响非常大，后来就增加了意图锁。所有的数据库(Oracle、Informix和Sybase)都有意图锁的实现机制。

IN 锁用于表以允许未提交读。

除了表锁之外，DB2 还支持表 5-4 所示的几种方式的行锁。

表 5-4 DB2 支持的行锁

名称缩写	全　名	需要表锁的最低级别	描　述
S	共享(Share)锁	IS	该行正被某个应用程序读取，其他应用程序只能对该行进行读操作
U	更改(Update)锁	IX	某个应用程序正在读该行并有可能更改该行，其他应用程序只能读该行
X	排它(eXclusive)锁	IX	该行正被某个应用程序更改，其他应用程序不能访问该行
W	弱排它 (Weak eXclusive)锁	IX	当一行数据被插入表中的时候，该行会被加上 W 锁。锁的拥有者能够更改该行，该锁与 X 锁基本相同，但它与 NW 锁兼容
NS	下一键共享 (Next Key Share)锁	IS	锁的拥有者和其他程序都可以读该行，但不能对该行进行更改。当应用程序处于 RS 或 CS 隔离级别时，该锁可用来替代 S 锁
NX	下一键排它 (Next Key eXclusive)锁	IX	当一行数据被插入索引中或从索引中被删除的时候，该行的下一行会被加上该锁。锁的拥有者可以读，但不能更改锁定行。该锁与 X 锁类似，但它与 NS 锁兼容
NW	下一键弱排它 (Next Key Weak eXclusive)锁	IX	当一行被插入索引中的时候，该行的下一行会被加上该锁。锁的拥有者可以读但不能更改锁定行。该锁与 X 和 NX 锁类似，但它与 W 和 NS 锁兼容

5.2.4 如何获取锁

大多数情况下，DB2 数据库管理程序在需要锁时会隐式地获取它们，因此这些锁在 DB2 数据库管理程序的控制之下。除了使用未提交读隔离级别的情况外，事务从不需要显式地请求锁。事实上，唯一有可能被事务显式锁定的数据库对象是表(LOCK TABLE)。

图 5-5 说明了如何确定所引用的对象获取的锁的类型。

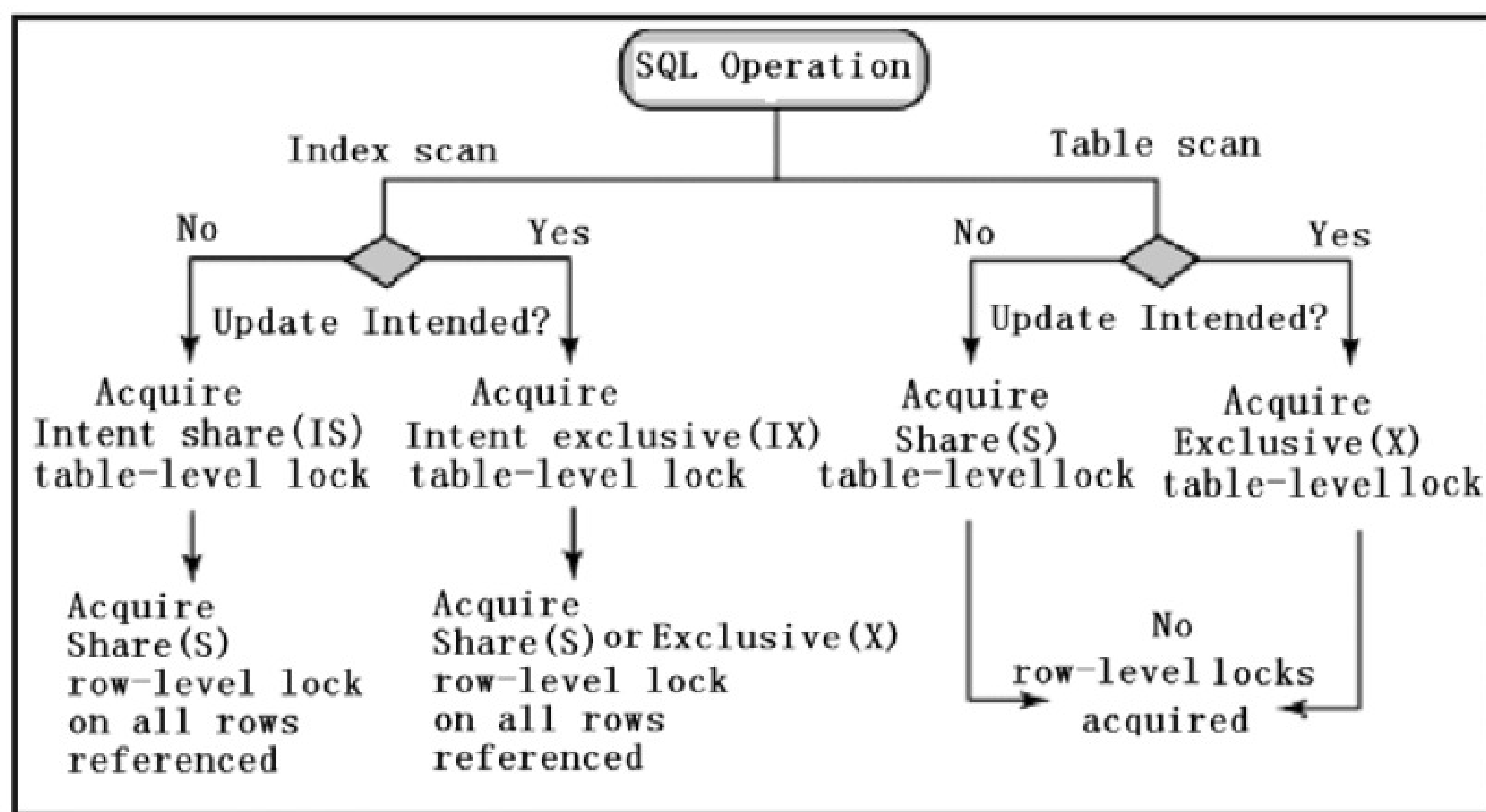


图 5-5　如何确定获取锁的类型

DB2 数据库管理程序总是尝试获取行级锁。但是，可以通过执行特殊形式的 ALTER TABLE 语句来修改这种行为，如下所示：

```
ALTER TABLE [TableName] LOCKSIZE TABLE
```

其中的 *TableName* 标识现有表的名称，所有事务在访问它时都要获取表级锁。

也可以通过执行 LOCK TABLE 语句，强制 DB2 数据库管理程序为特定事务在表上获取表级锁，如下所示：

```
LOCK TABLE [TableName] IN [SHARE | EXCLUSIVE] MODE
```

其中的 TableName 标识现有表的名称，对于这个表应该获取表级锁(假定其他事务在该表上没有不兼容的锁)。如果在执行这条语句时指定了共享(SHARE)模式，就会获得允许其他事务读取(但不能更改)存储在表中的数据的表级锁；如果执行时指定了互斥(EXCLUSIVE)模式，就会获得不允许其他事务读取或修改存储在表中的数据的表级锁。

ALTER TABLE 语句的 LOCKSIZE 子句指定行级别或表级别的锁定的作用域(详细程度)。默认情况下使用的是行锁定。由这些已定义的表锁定仅请求 S(共享)和 X(互斥)锁定。ALTER TABLE 语句的 LOCKSIZE ROW 子句不会阻止正常的锁定升级。

在下列情况下，由 ALTER TABLE 语句定义的永久表锁定可能比使用 LOCK TABLE 语句获得的单个事务表锁定更可取：

- 表是只读的，并且始终只需要 S 锁定。其他用户也可以获取表的 S 锁定。
- 表通常由只读应用程序访问，但有时也需要单个用户访问以进行简单维护，该用户需要 X 锁定。当维护程序运行时，将只读应用程序锁定在外，但在其他情况下，只读应用程序可以在使用最小锁定开销的同时访问表。

ALTER TABLE 语句在全局指定锁定，会影响访问该表的所有应用程序和用户。单个应用程序可以使用 LOCK TABLE 语句来指定应用程序级别的表锁定。

5.2.5 锁的兼容性

在一个应用程序已经对某个对象锁定的情况下，另一个应用程序请求对同一对象的锁定，此时就会出现锁定兼容性问题。当两种锁定方式兼容时，可以同意对该对象的第二个锁定请求。如果请求的锁定方式与已挂起的锁定方式不兼容，那么就不能同意第二个锁定请求。相反，请求要等到第一个应用程序释放锁定，并且释放所有其他现有的不兼容锁定为止。

表 5-5 和表 5-6 显示了锁对象兼容的有关信息。在某些状态下，当另一个进程挂起或正在请求对同一资源的锁定时，可以同意锁定请求。否则请求者就必须等待，直到所有不兼容的锁定被其他进程释放为止。

注意，当请求者正等待锁定时，可能出现超时。

表 5-5 表锁的兼容性

锁 A 的方式	锁 B 的方式							
	IN	IS	S	IX	SIX	U	X	Z
IN	兼容	兼容	兼容	兼容	兼容	兼容	兼容	不兼容
IS	兼容	兼容	兼容	兼容	兼容	兼容	不兼容	不兼容
S	兼容	兼容	兼容	不兼容	不兼容	兼容	不兼容	不兼容
IX	兼容	兼容	不兼容	兼容	不兼容	不兼容	不兼容	不兼容
SIX	兼容	兼容	不兼容	不兼容	不兼容	不兼容	不兼容	不兼容
U	兼容	兼容	兼容	不兼容	不兼容	不兼容	不兼容	不兼容
X	兼容	不兼容	不兼容	不兼容	不兼容	不兼容	不兼容	不兼容
Z	不兼容	不兼容	不兼容	不兼容	不兼容	不兼容	不兼容	不兼容

表 5-6　行锁的兼容性

锁 A 的模式	锁 B 的模式						
	S	U	X	W	NS	NX	NW
S	兼容	兼容	不兼容	不兼容	兼容	不兼容	不兼容
U	兼容	不兼容	不兼容	不兼容	兼容	不兼容	不兼容
X	不兼容	不兼容	不兼容	不兼容	不兼容	不兼容	不兼容
W	不兼容	不兼容	不兼容	不兼容	不兼容	不兼容	兼容
NS	兼容	兼容	不兼容	不兼容	兼容	兼容	兼容
NX	不兼容	不兼容	不兼容	不兼容	兼容	不兼容	不兼容
NW	不兼容	不兼容	不兼容	兼容	兼容	不兼容	不兼容

5.3　隔离级别(Isolation Levels)

维护数据库的一致性和数据完整性，同时又允许多个应用程序同时访问同一数据，这样的特性称为并发性。DB2 数据库用来尝试强制实施并发性的方法之一是通过使用隔离级别，隔离级别决定在第一个事务访问数据时，如何对其他事务锁定或隔离该事务使用的数据。DB2 使用下列隔离级别来强制实施并发性：

- 可重复读(Repeatable Read，RR)
- 读稳定性(Read Stability，RS)
- 游标稳定性(Cursor Stability，CS)
- 未提交读(Uncommitted Read，UR)

可重复读隔离级别可以防止所有现象，但是会大大降低并发性(可以同时访问同一资源的事务数量)。未提交读隔离级别提供了最大的并发性，但是“脏读”、“幻像读”和“不可重复读”都可能出现。默认的隔离级别是 CS。

5.3.1　可重复读(RR—Repeatable Read)

可重复读隔离级别是最严格的隔离级别。在该隔离级别下，事务的影响完全与其他并发事务隔离，脏读、不可重复读、幻像读现象都不会发生。当使用可重复读隔离级别时，在事务执行期间会锁定该事务以任何方式引用的所有行。因此，如果在同一事务中发出同一条 SELECT 语句两次或更多次，那么产生的结果数据集总是相同的。使用可重复读隔离级别的事务可以多次检索同一行集，并对它们执行任意操作，直到提交或回滚操作终止该事务为止。但是，在事务存在期间，不允许其他事务执行会影响这个事务正在访问的任何行的插入、更新或删除操作。为了确保这种行为不会发生，锁定该事务引

用的每一行——而不是仅锁定被实际检索或修改的那些行。因此，如果事务扫描了 1000 行，但只检索 10 行，那么事务扫描的 1000 行(而不仅是被检索的 10 行)都会被锁定。

那么在现实环境中可重复读隔离级别是如何工作的呢？假定如家酒店使用 DB2 数据库跟踪的客房信息，包括房间预订和房价信息，还有一个基于 Web 的应用程序，它允许顾客按“先到先服务”的原则预订房间。如果旅馆预订应用程序是在可重复读隔离级别下运行的，当顾客扫描某个段日期内的可用房间列表时，你(旅馆经理)将无法更改那些房间在指定日期范围内的房价。同样，其他顾客也无法进行或取消将会更改该列表的预订操作(直到第一个顾客的事务终止为止)。图 5-6 说明了这种行为。

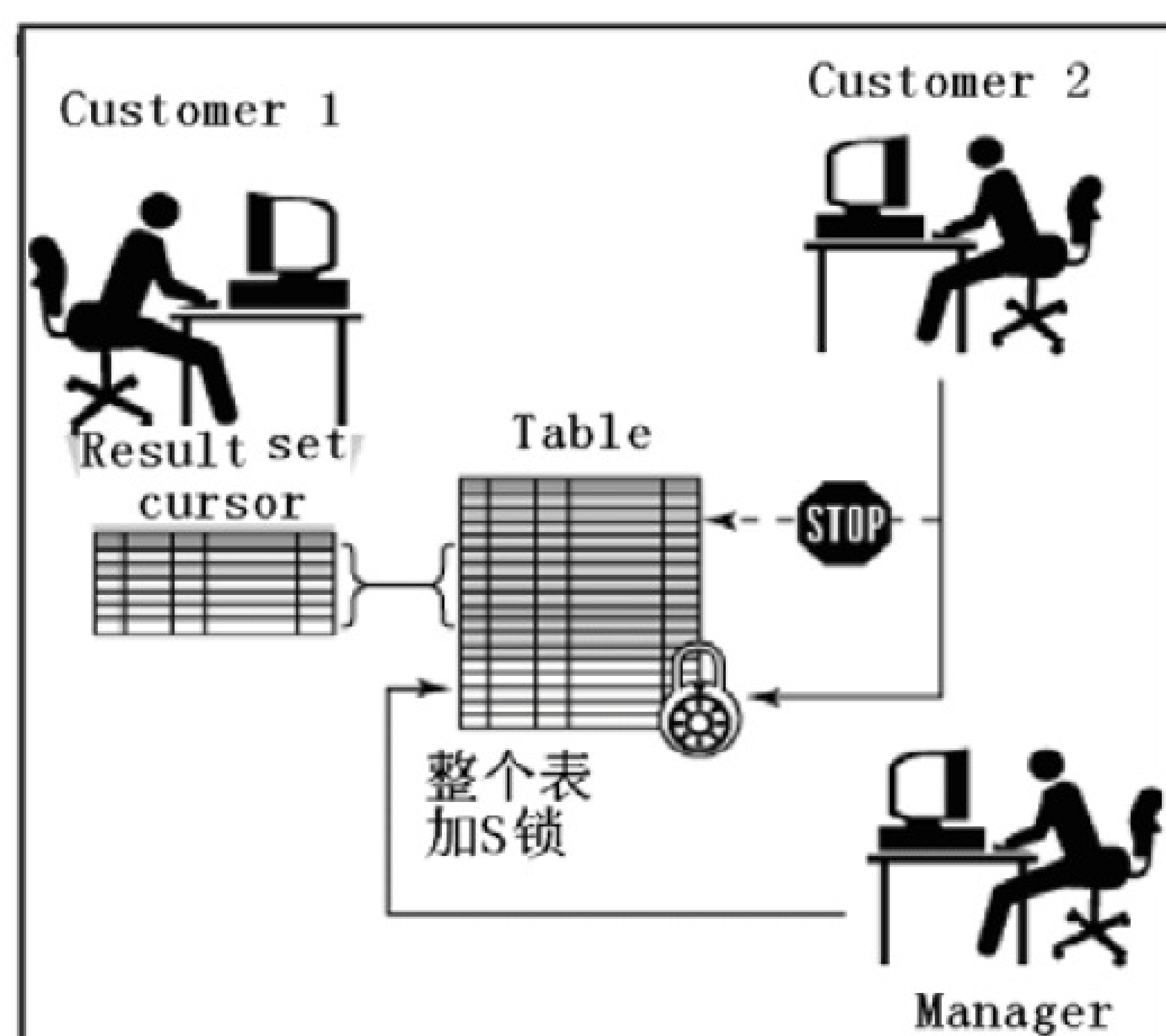


图 5-6　可重复读隔离级别的示例

在使用可重复读隔离级别时，事务中所有被读取过的行都会被加上 S 锁，直到事务被提交或回滚，行上的锁才会被释放。这样可以保证在事务中即使多次读取同一行，得到的值也不会改变。另外，在同一事务中即使以同样的搜索标准重新打开已被处理过的游标，得到的结果集也不会改变。可重复读相对于读稳定性而言，加锁的范围更大。对于读稳定性，应用程序只对符合要求的所有行加锁；而对于可重复读，应用程序将对所有被扫描过的行加锁。

可重复读(RR)会锁定应用程序在工作单元中引用的所有行。利用“可重复读”，在打开游标的相同工作单元内，由应用程序发出 SELECT 语句两次，每次都返回相同的结果。利用“可重复读”，不可能出现丢失更新、脏读和幻像读的情况。

在工作单元完成之前，“可重复读”应用程序可以尽可能多次地检索和操作这些行。但是，在工作单元完成之前，其他应用程序均不能更新、删除或插入可能会影响结果表的行。“可重复读”应用程序不能查看其他应用程序的未提交的更改。

5.3.2　读稳定性(RS—Read Stability)

读稳定性隔离级别没有可重复读隔离级别那么严格；因此，它没有将事务与其他并发事务的效果完全隔离。读稳定性隔离级别可以防止脏读和不可重复读，但是可能出现幻像读。在使用这个隔离级别时，只是锁定事务实际检索和修改的行。因此，如果事务扫描了1000 行，但只检索 10 行，那么只有被检索的 10 行(而不是所扫描的 1000 行)被锁定。因此，如果在同一事务中发出同一条 SELECT 语句两次或更多次，那么每次产生的结果数据集可能不同。

与可重复读隔离级别一样，在读稳定性隔离级别下运行的事务可以检索行集，并可以对它们执行任意操作，直到事务终止。在这个事务存在期间，其他事务不能执行那些会影响这个事务检索到的行集的更新或删除操作；但是其他事务可以执行插入操作。如果插入的行与第一个事务的查询的选择条件匹配，那么这些行可能作为幻像出现在后续产生的结果数据集中。其他事务对其他行所做的更改，在提交之前是不可见的。

那么，读稳定性隔离级别会如何影响如家酒店客房预定应用程序的工作方式呢？当一个顾客检索某段日期内的所有可用房间列表时，可以更改这个顾客的列表之外的任何房间的房价。同样，其他顾客可以进行或取消房间预订。如果第一个顾客再次运行同样的查询，其他顾客的操作可能会影响第一个顾客获得的可用房间列表。如果第一个顾客再次查询同一段日期内的所有可用房间列表，产生的列表中有可能包含新的房价或第一次产生列表时不可预订的房间。图 5-7 说明了这种行为。

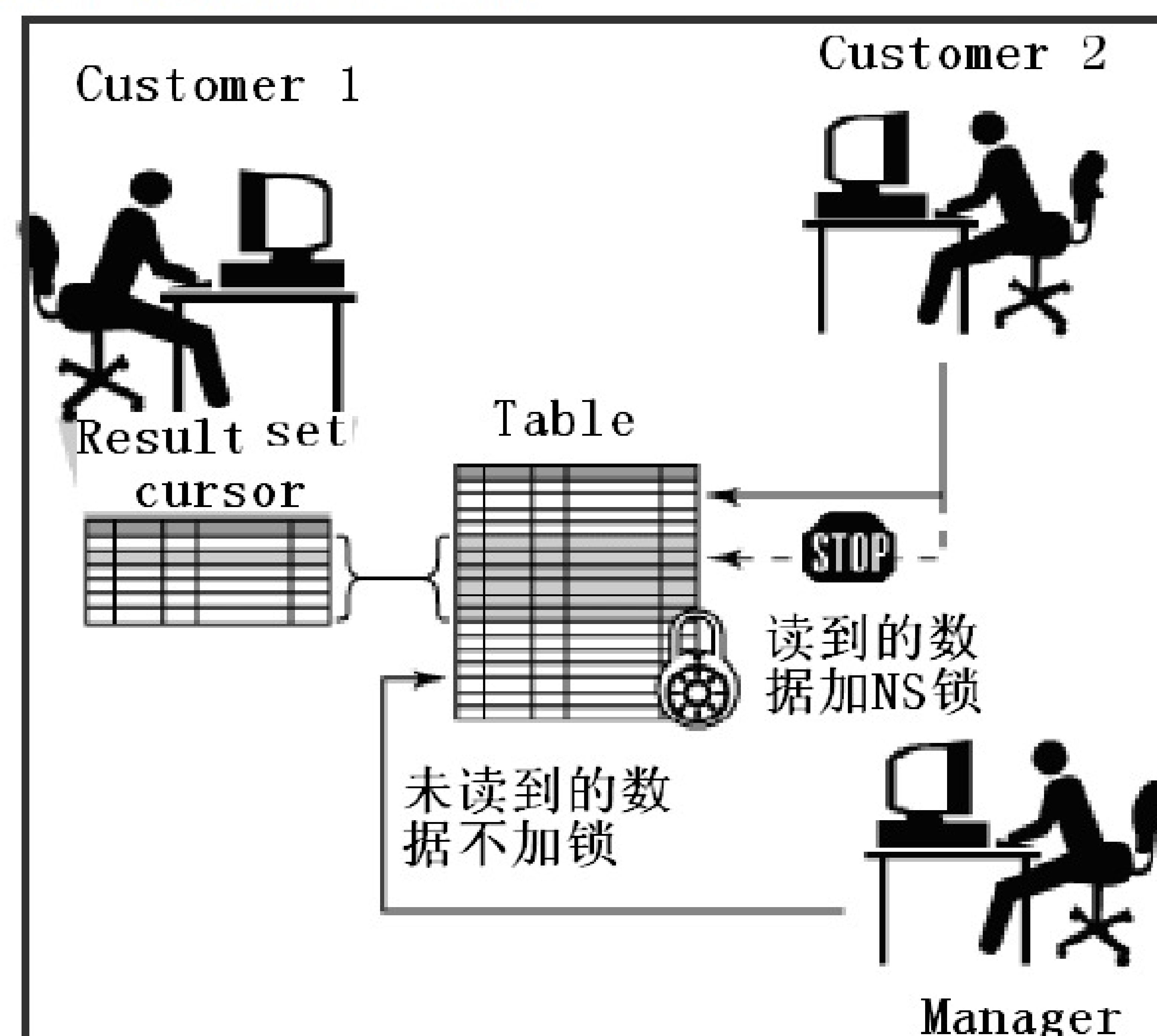


图 5-7　读稳定性隔离级别的示例

在使用读稳定性隔离级别时，事务中所有被读取过的行都会被加上 NS 锁，直到事务被提交或回滚，行上的锁才会被释放。这样可以保证在事务中即使多次读取同一行，得到的值也不会改变。但是，如果使用这种隔离级别，那么在事务中，如果使用同样的搜索标准重新打开已被处理过的游标，结果集可能改变(可能会增加某些行，这些行被称为幻影行(Phantom)。这是因为 RS 隔离级别不能阻止通过插入或更新操作在结果集中加入新行)。

读稳定性(RS)只锁定应用程序在工作单元中检索的那些行，确保在工作单元完成之前，在工作单元运行期间的任何限定行读取不被其他应用程序进程更改，并且确保不会读取由另一个应用程序进程更改的任何行，直至该进程落实了这些更改为止。也就是说，不可能出现“不可重复读”的情形。

“读稳定性”隔离级别的目标之一是提供较高并行性以及数据的稳定视图。为了有助于达到此目标，优化器确保在发生锁定升级前不获取表级锁定。

“读稳定性”隔离级别最适用于包括下列所有特征的应用程序：

- 在并发环境下运行
- 必须限定某些行在工作单元运行期间保持稳定
- 在工作单元中不会多次发出相同的查询，或者在同一工作单元中发出多次查询时并不要求查询获得相同的回答

5.3.3 游标稳定性(CS—Cursor Stability)

游标稳定性隔离级别在隔离事务效果方面非常宽松，可以防止脏读，但有可能出现不可重复读和幻像读。这是因为在大多数情况下，游标稳定性隔离级别只锁定由事务声明并打开的游标当前引用的行。

当使用游标稳定性隔离级别的事务通过游标从表中检索行时，其他事务不能更新或删除游标引用的行。但是，如果被锁定的行本身不是用索引访问的，那么其他事务可以将新的行添加到表中，以及对被锁定行前后的行进行更新和/或删除操作。所获取的锁一直有效，直到游标重定位或事务终止为止(如果游标重定位，原来行上的锁就被释放，并获得游标现在所引用行上的锁)。此外，如果事务修改检索到的任何行，那么在事务终止之前，其他事务不能更新或删除该行，即使游标不再位于被修改的行。与可重复读和读稳定性隔离级别一样，其他事务在其他行上进行的更改，在这些更改提交之前对于使用游标稳定性隔离级别(这是默认的隔离级别)的事务是不可见的。

如果如家酒店的客房预订应用程序在游标稳定性隔离级别下运行，那么会有什么影响呢？当顾客检索某段日期内所有可用房间的列表，然后查看产生的列表中每个房间的信息

时(每次查看一个房间)，可以更改旅馆中任何房间的房价，但是顾客当前正在查看的房间除外(对于指定的日期范围)。同样，其他顾客可以对任何房间进行或取消预订，但是第一个顾客当前正在查看的房间除外(对于指定的日期范围)。也就是说，对于第一个顾客当前正在查看的房间记录，你和其他顾客都不能进行任何操作。当第一个顾客查看列表中另一个房间的信息时，你和其他顾客就可以修改他刚才查看的房间记录(如果这个顾客没有预订这个房间的话)。图 5-8 说明了这种行为。

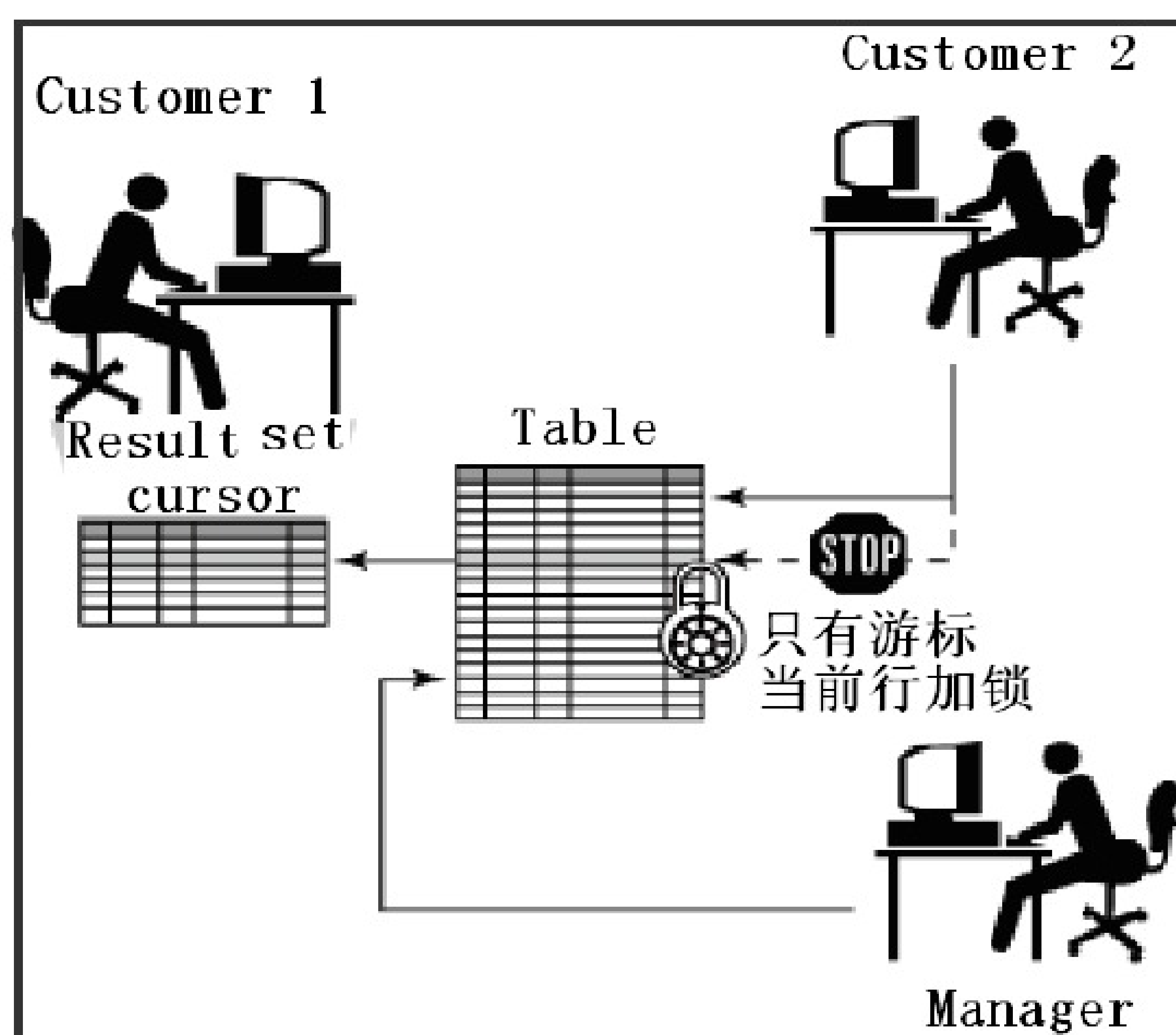


图 5-8 游标稳定性隔离级别的示例

在使用游标稳定性隔离级别时，事务的结果集中只有正在被读取的那一行(游标指向的行)会被加上 NS 锁，其他未被处理的行不被加锁。这种隔离级别只能保证正在被处理的行的值不会被其他并发的程序所改变。该隔离级别是 DB2 默认的隔离级别。

当在行上定位游标时，游标稳定性(CS)会锁定任何由应用程序的事务访问的行。此锁定在读取下一行或终止事务之前有效。但是，如果更改某一行上的任何数据，那么在对数据库落实更改之前必须挂起该锁定。

对于具有“游标稳定性”的应用程序已检索的行，当该行上有任何可更新的游标时，任何其他应用程序都不能更新或删除该行。“游标稳定性”应用程序不能查看其他应用程序的未落实更改。

使用“游标稳定性”隔离级别，可能会出现不可重复读和幻像读现象。“游标稳定性”是默认隔离级别，建议在需要最大并行性、但只看到其他应用程序中已落实行的情况下才使用。

5.3.4 当前已提交(Currently Committed)

但是在 CS 隔离级别下，在查询过程中，DB2 会对游标所在的行加 NS 锁。如果该行目前正在被修改，那么也会堵塞查询。随着互联网的不断发展，基于 Internet 的非账务类应用往往具有大并发、高响应的特点。对于这些应用，频繁的锁等待是无法容忍的，因此在 DB2 V9.7 的 CS 隔离级别里，新增了名为当前已提交(Currently Committed)的数据库配置选项。在默认当前已提交启用的情况下，DB2 在查询的时候不会再被修改的行阻塞，如果发现未提交的变化行数据，那么将使用当前已提交的数据。此行为基于日志，不需要额外的维护操作，可以有效避免超时和死锁现象。

可以通过修改 db cfg 来启动或关闭当前提交：

```
$db2 update db cfg for test using cur_commit on
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
```

比如在上面的酒店预订例子中，如果第一个顾客正在预订某个房间，在不使用当前已提交的情况下，其他顾客需要等待第一个顾客预订完才能读取该房间的信息；但是如果启用当前已提交，那么其他顾客在查询该房间的时候不需要等待，但是查询到的信息仍然是未修改前的信息。

5.3.5 未提交读(UR—Uncommitted Read)

未提交读隔离级别是最不严格的隔离级别。实际上，在使用这种隔离级别时，仅当另一个事务试图删除或更改被检索的行所在的表时，才会锁定被检索的行。因为在使用这种隔离级别时，行通常保持未锁定状态，所以脏读、不可重复读和幻像读都可能会发生。因此，未提交读隔离级别通常用于那些访问只读表和视图的事务，以及某些执行 SELECT 语句的事务(只要其他事务的未提交数据对这些语句没有负面效果)。

顾名思义，其他事务对行所做的更改在被提交之前对于使用未提交读隔离级别的事务是可见的。但是，此类事务不能看见或访问其他事务创建的表、视图或索引，直到那些事务被提交为止。类似地，如果其他事务删除了现有的表、视图或索引，那么仅当进行删除操作的事务终止时，使用未提交读隔离级别的事务才能知道这些对象不再存在了(一定要注意：当运行在未提交读隔离级别下的事务使用可更新游标时，该事务的行为和在游标稳定性隔离级别下运行一样，并应用游标稳定性隔离级别的约束)。

那么未提交读隔离级别对如家酒店的客房预订应用程序有什么影响呢？现在，当顾客检索某段日期内的所有可用房间列表时，你可以更改旅馆中任何房间在任何日期的房价，而其他顾客也可以对任何房间进行或取消预订，包括第一个顾客当前正在查看的房间记录(对于指定的日期范围)。另外，第一个顾客生成的房间列表可能包含其他顾客正在预订(因

此实际上不可用)的房间。图 5-9 说明了这种行为。

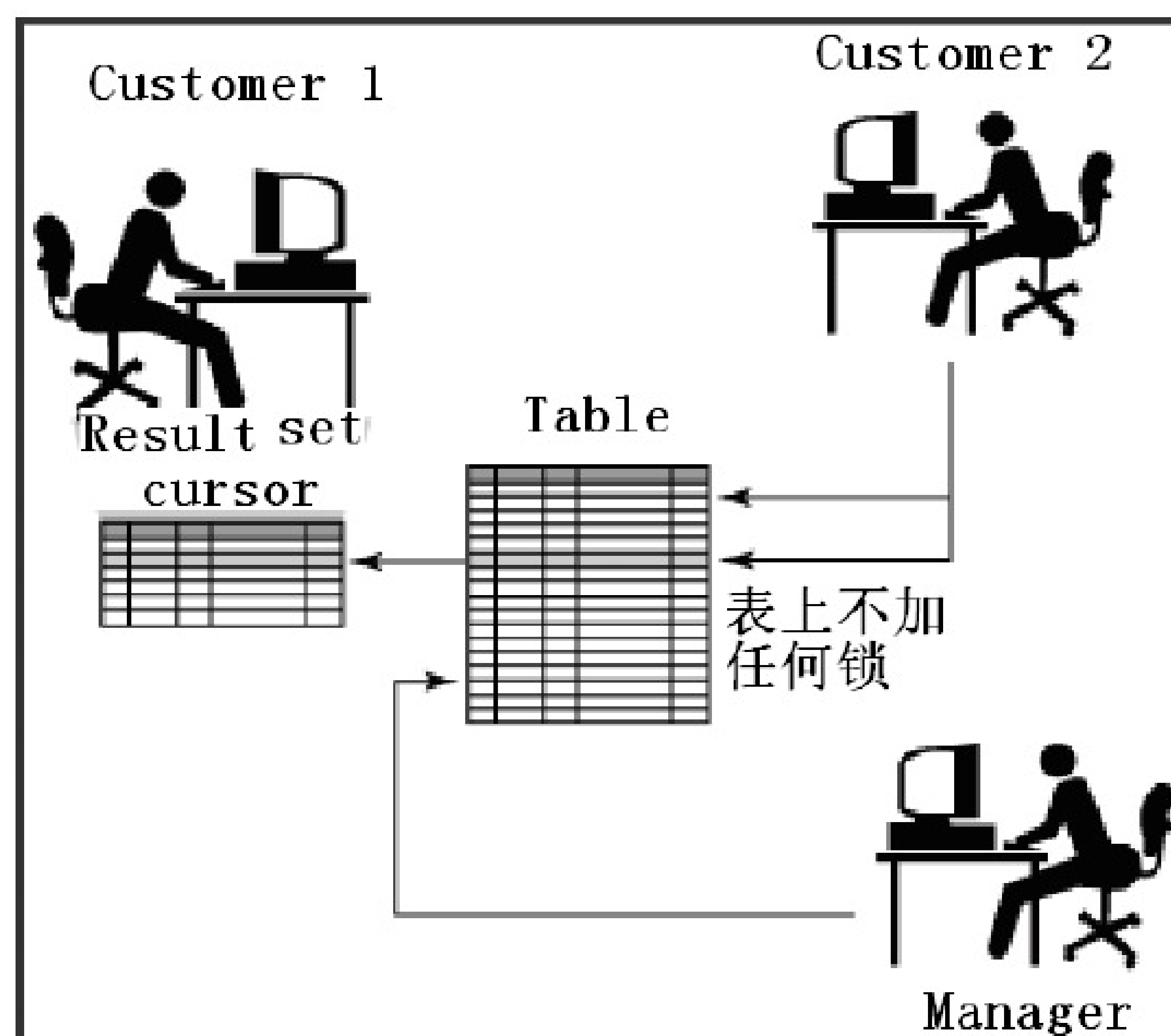


图 5-9　未提交读隔离级别的示例

未提交读(UR)允许应用程序访问其他事务的未提交更改。除非其他应用程序尝试删除或改变该表，否则该应用程序不会锁定正读取的行而使其他应用程序不能访问该行。对于只读和可更新的游标，“未提交读”的工作方式有所不同。

在使用未提交读隔离级别时，对于只读操作，不加行锁。典型的只读操作包括 SELECT 语句的结果集(比如语句中包括 ORDER BY 子句)、定义游标时指明 FOR FETCH ONLY。

未提交读隔离级别可以改善应用程序的性能，同时可以最大程度地允许并发。但是，应用程序的数据完整性将受到威胁。如果需要读取未提交的数据，该隔离级别是唯一选择。

只读游标可以访问大多数其他事务未落实的更改。但是，当该事务正在处理时，不能使用正由其他事务创建或删除的表、视图和索引。其他事务的任何其他更改在落实或回滚前都可被读取。

注意：

“未提交读”隔离级别下的可更新操作的游标的工作方式与游标稳定性隔离级别下的相同。

当使用隔离级别 UR 运行程序时，应用程序可以使用隔离级别 CS。发生这种情况的原因是应用程序中使用的游标是模糊游标。利用 BLOCKING 选项，可以将模糊游标升级为隔离级别 CS。BLOCKING 选项的默认值是 UNAMBIG。这意味着模糊游标是可更新的，并且隔离级别可以升级为 CS。要防止此升级，有两种选择：

- 修改应用程序中的游标为非模糊游标。将 SELECT 语句更改为包括 FOR READ ONLY 子句。
- 将模糊游标保留在应用程序中，但是使用预编译程序或 BLOCKING ALL 和 STATICREADONLY YES 选项绑定它，以允许在运行应用程序时将任何模糊游标视为只读游标。

同对扫描 1000 行的“可重复读”给出的示例一样，如果使用“未提交读”，那么不需要任何行锁定。

使用“未提交读”，可能出现不可重复读和幻像读现象。“未提交读”隔离级别最常用于只读表上的查询。如果仅执行选择语句且不关心是否可从其他应用程序中看到未提交的数据，那么该隔离级别也是不错的选择。

以上所讲的隔离级别的加锁范围和持续时间都是针对读操作而言的。对于更改操作，被修改的行会被加上 X 锁，无论使用何种隔离级别，X 锁都直到提交或回滚之后才会被释放。

5.3.6 隔离级别的摘要

表 5-7 按不期望的结果概述了几个不同的隔离级别。

表 5-7 隔离级别的摘要

隔 离 级 别	访问未落实的数据	不可重复读	幻 像 读
可重复读(RR)	不可能	不可能	不可能
读稳定性(RS)	不可能	不可能	可能
游标稳定性(CS)	不可能	可能	可能
未提交读(UR)	可能	可能	可能

表 5-8 提供了简单的试探方法，以帮助你为应用程序选择初始隔离级别。首先考虑表 5-8 中所示的方法，并参阅先前对各隔离级别的讨论，以找到最适合的隔离级别。

表 5-8 选择隔离级别的准则

应用程序类型	需要高数据稳定性	不需要高数据稳定性
读写事务	RS	CS
只读事务	RR 或 RS	UR

为应用程序选择适当的隔离级别对于应用程序避免无法容忍的现象很重要。因为获取和释放锁定所需的 CPU 和内存资源会随隔离级别的不同而不同，所以隔离级别不但影响应用程序之间的隔离程度，而且还影响应用程序的性能特征。潜在的锁等待情况也会随隔离级别的不同而不同。

因为隔离级别确定访问数据时如何锁定数据并使数据不受其他进程影响，所以应该选择能平衡并行性和数据完整性需求的隔离级别。指定的隔离级别在工作单元运行期间生效。

1. 选择正确的隔离级别

使用的隔离级别不仅影响数据库对并发性的支持程度，而且影响并发应用程序的性能。通常，使用的隔离级别越严格，并发性就越小，某些应用程序的性能还可能会越低，因为它们要等待资源上的锁被释放。那么，如何决定要使用哪种隔离级别呢？最好的方法是确定哪些现象是不可接受的，然后选择能够防止这些现象发生的隔离级别：

- 如果正在执行大型查询，而且不希望并发事务所做的修改会导致查询的多次运行而返回不同的结果，那么使用可重复读隔离级别。
- 如果希望在应用程序之间获得一定的并发性，并且希望限定的行在事务执行期间保持稳定，那么使用读稳定性隔离级别。
- 如果希望获得最大的并发性，同时不希望查询看到未提交的数据，那么使用游标稳定性隔离级别。
- 如果正在只读的表/视图/数据库上执行查询，或者并不介意查询是否返回未提交的数据，那么使用未提交读隔离级别。

2. 指定要使用的隔离级别

尽管隔离级别控制事务级的行为，但实际上是在应用程序级被指定的：

- 对于嵌入式 SQL 应用程序，在预编译时或在将应用程序绑定到数据库(如果使用延迟绑定)时指定隔离级别。在这种情况下，使用 PRECOMPILE 或 BIND 命令的 ISOLATION 选项来设置隔离级别。
- 对于开放数据库连接(Open Database Connectivity，ODBC)和调用级接口(Call Level Interface， CLI)应用程序，隔离级别是在应用程序运行时通过调用指定了 SQL_ATTR_TXN_ISOLATION 连接属性 SQLSetConnectAttr()函数进行设置的(另外，也可以通过指定 db2cli.ini 配置文件中 TXNISOLATION 关键字的值来设置 ODBC/CLI 应用程序的隔离级别。但是，这种方法不够灵活，不能像第一种方法那样为应用程序中的不同事务修改隔离级别)。
- 对于 Java 数据库连接(Java Database Connectivity，JDBC)和 SQLJ 应用程序，隔离级别是在应用程序运行时通过调用 DB2 的 java.sql 连接接口中的 setTransactionIsolation()方法设置的。

当没有使用这些方法显式指定应用程序的隔离级别时，默认使用游标稳定性隔离级别。这个默认设置适用于从命令行处理程序(CLP)执行的 DB2 命令、SQL 语句和脚本，以及嵌入式 SQL、ODBC/CLI、JDBC 和 SQLJ 应用程序。因此，也可以为从 CLP 执行的操作(以及传递给 DB2 CLP 进行处理的脚本)指定隔离级别。在这种情况下，隔离级别是通过在建立数据库连接之前在 CLP 中执行 CHANGE ISOLATION 命令设置的：

```
$db2 change isolation to ur
DB21027E  当连接至数据库时未能更改隔离级别
$db2 connect reset
DB20000I  SQL 命令成功完成
$db2 change isolation to ur
DB21053W  当连接至不支持 UR 的数据库时，会发生自动升级
DB20000I  CHANGE ISOLATION 命令成功完成
```

在 DB2 UDB 7.1 及更高版本中，能够指定特定查询所用的隔离级别，方法是在 SELECT SQL 语句中加上 WITH [RR | RS | CS | UR]子句。使用这个子句的简单 SELECT 语句示例如下所示：

```
SELECT * FROM EMPLOYEE WHERE EMPID = '001' WITH RR
```

如果应用程序在大多数时候需要比较宽松的隔离级别(以支持最大的并发性)，但是对于其中的某些查询必须防止某些现象出现，那么这条子句就是帮助你实现目标的好方法。

5.4 锁转换、锁等待、锁升级和死锁

5.4.1 锁转换及调整案例

更改已挂起的锁定方式被称为转换。当进程访问已挂起锁定的数据对象，且访问方式需要比已挂起的锁定更严格时，数据库管理器将数据对象上现有的锁模式与被请求的锁模式进行比较，如果需要的锁模式更高，将进行锁转换。进程在任意时间对数据对象只能挂起锁定，尽管可以通过查询间接地对同一数据对象多次请求锁定。在事务中，当对象需要不同的加锁模式时，为对象加上更高模式的锁，否则将保持现有的锁模式。锁模式由高到低按照以下顺序排列：

表锁：IN→IS→S→IX→U→X→Z

行锁：S→U→X

其中 S 锁与 IX 锁的转换比较特殊，当应用程序拥有表上的 S 锁并请求 IX 锁的时候，锁转换的结果为 SIX 锁。或者，当应用程序拥有表上的 IX 锁并请求 S 锁的时候，锁转换的结果也是 SIX 锁。

某些锁定方式仅适用于表，而有些锁定方式仅适用于行。对于行，如果需要 X 锁且挂起 S 或 U(更新)锁，那么进行转换。

但是，在锁定转换这一点上，IX(意向互斥)和 S(共享)锁是特殊情况。S 和 IX 锁定的严格程度相当，所以如果挂起了一个而请求另一个，那么结果转换为 SIX(带意向互斥的共享)锁定。如果请求的方式更严格，那么所有其他的转换都将导致请求的锁定方式变成挂起的锁定方式。

当查询更新行时，也可以发生双重转换。如果行是通过索引访问读取的且锁定为 S，那么包含该行的表具有覆盖意向锁定。但是，如果锁定类型是 IS 而不是 IX，并且随后更改了行，那么表锁定将转换为 IX，而行锁定转换为 X。

注意：

在执行 SQL 时，锁定转换通常隐式地进行，由数据库自动完成而不需要用户参与。了解不同的查询以及表和索引的组合能获得的锁定种类，有助于设计和调整应用程序。

表 5-9 是一个锁转换的例子，这里假设当前应用程序的隔离级别为 RR，IXCOL=?谓词采用的扫描方式为索引扫描。

表 5-9 锁转换示例

语　句	语句所需的表锁	是否需要锁转换	结　果　锁	说　明
SELECT … FROM A WHERE IXCOL=?	IS	不需要	IS	该事务中的第一条语句，不会发生锁转换
SELECT … FROM A	S	需要	S	S 锁的模式比 IS 锁高，需要锁转换
SELECT … FROM A WHERE IXCOL=?	IS	不需要	IS	IS 锁的模式比 S 锁低，所以不需要锁转换
UPDATE A SET… WHERE IXCOL=?	IX	需要	SIX	原先是 S 锁，现在又申请 IX 锁，锁转换的结果为 SIX 锁
COMMIT	不需要锁	不需要	所有锁被释放	在提交时锁被释放
SELECT … FROM A WHERE IXCOL=?	IS	不需要	IS	新事务的开始
LOCK TABLE A IN EXCLUSIVE MODE	X	需要	X	X 锁的模式比 IS 锁高，需要进行锁转换
UPDATE A SET… WHERE IXCOL=?	IX	不需要	X	IX 锁的模式比 X 锁低，不需要锁转换
ROLLBACK	不需要锁	不需要	所有锁被释放	在回滚时锁被释放

注意：

所有的锁在提交或回滚时都会被释放，除了定义游标时与指定为 WITH HOLD 的游标相关的表锁。

5.4.2 锁升级及调整案例

数据库管理器可以自动将锁定从行级别升级为表级别。对于分区表，数据库管理器可以自动将锁定从行或块级别升级为数据分区级别。*maxlocks* 数据库配置参数用于指定触发锁定升级的百分比。获取触发锁定升级的锁定的表可能不受影响。每个锁在内存中都需要一定的内存空间，为了减少锁需要的内存开销，DB2 提供了锁升级这一功能。锁升级是通过对表加上非意图性的表锁，同时释放行锁来减少锁的数目，从而达到减少锁需要的内存开销的目的。锁升级由数据库管理器自动完成，数据库的配置参数锁列表页面数(locklist)和应用程序占有百分比(maxlocks)直接影响锁升级的处理，如图 5-10 所示。

在图 5-10 中，上图说明应用程序超出允许它具有的总的锁内存百分比(maxlocks 值)的情况。因此，数据库执行锁升级，将应用程序的锁内存百分比降低到 maxlocks 指定的百分比之下。下图说明了更为常见的情况——到达总的锁内存限制，因此执行锁升级来释放锁内存。

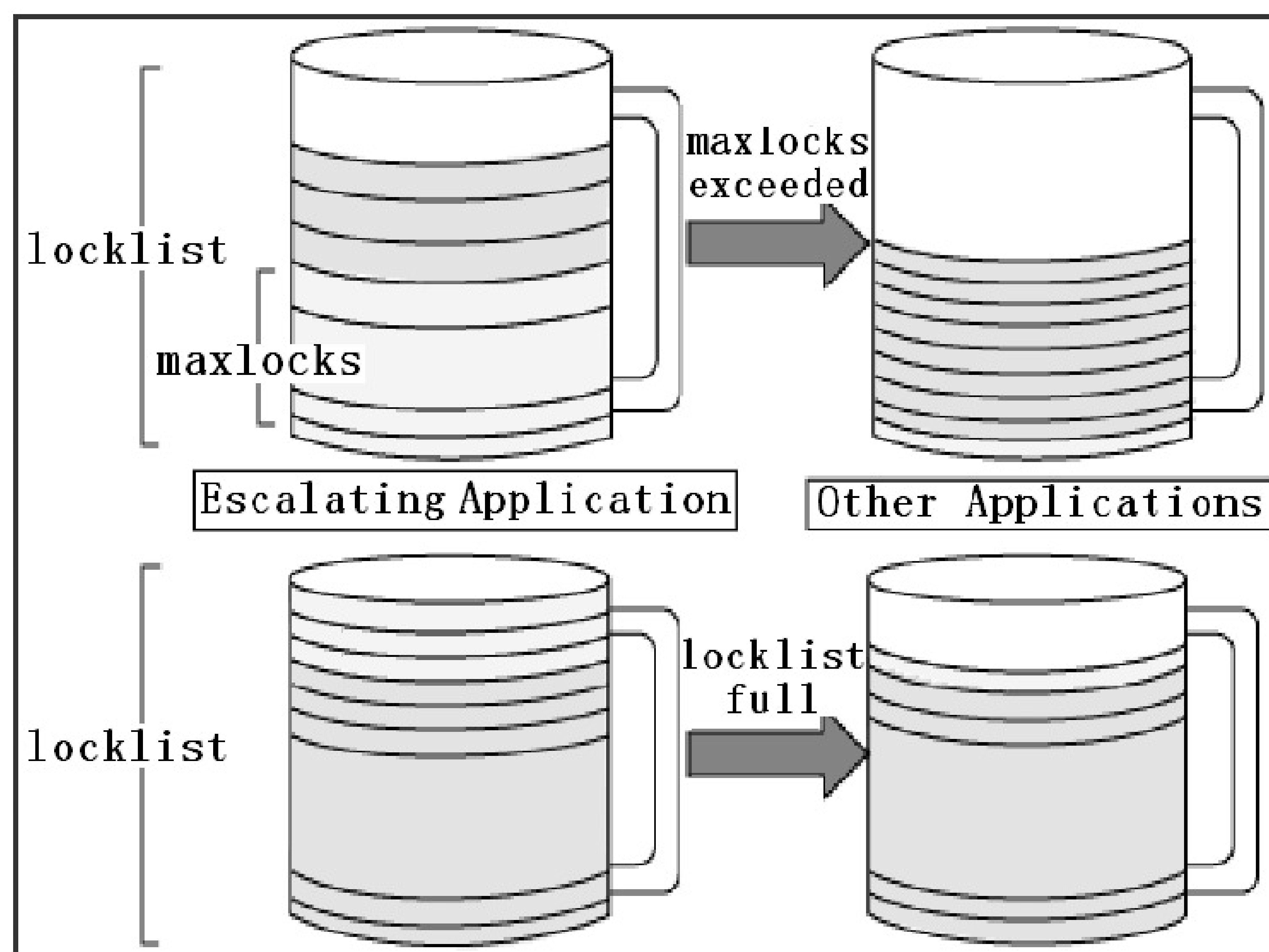


图 5-10 锁升级示意图

锁升级问题可以通过增加 locklist 和 maxlocks 数据库参数的大小来解决。但是，如果仍然遇到锁定问题，应检查是否因未能提交事务而未释放已更新行上的锁。

每个数据库都有锁列表，锁列表包含所有同时连接到数据库的应用程序所持有的锁。在 DB2 V9.7 64 位平台上，第一个锁要求占 256 字节，而其他锁要求占 128 字节。

当应用程序使用的 locklist 的百分比达到 maxlocks 时，数据库管理器将执行一次锁升级(lock escalation)，在这个操作中会将行锁转换成单独的表锁。而且，如果 locklist 快要耗尽，数据库管理器将找出持有表上最多行锁的连接，并将这些行锁转换成表锁以释放 locklist 内存。锁定整个表会大大降低并发性，锁等待或锁超时的几率也就增加了。

- locklist 表明分配给锁列表的存储容量。每个数据库都有锁列表，锁列表包含了并发连接到数据库的所有应用程序所持有的锁。锁定是数据库管理器用来控制多个应用程序并发访问数据库中数据的机制。行和表都可以被锁定。
- maxlocks 定义了应用程序持有的锁列表的百分比，在数据库管理器执行锁升级之前必须填充锁列表。当应用程序使用的锁列表百分比达到 maxlocks 时，数据库管理器会升级这些锁，这意味着用表锁代替行锁，从而减少列表中锁的数量。当任何应用程序持有的锁数量达到整个锁列表大小的这个百分比时，对应用程序持有的锁进行锁升级。如果锁列表用完了空间，那么也会发生锁升级。数据库管理器通过查看应用程序的锁列表并查找行锁最多的表，决定对哪些锁进行升级。如果用表锁替换这些行锁，将不再会超出 maxlocks 值，那么锁升级就会停止。否则，锁升级就会一直进行，直到持有的锁列表百分比低于 maxlocks。maxlocks 参数乘以 maxappls 参数的值不能小于 100。

locklist 配置参数的计算方法如下(操作系统为 DB2 V9.7 64 位平台)：

(1) 计算锁列表大小的下限：(512 * 128 *maxappls)/4096。其中，512 是每个应用程序平均所含锁数量的估计值，32 是对象(已有一把锁)上每把锁所需的字节数。

(2) 计算锁列表大小的上限：(512 * 256 *maxappls)/4096。其中，64 是某个对象上第一把锁所需的字节数。

(3) 对于数据，估计可能具有的并发数，并根据预计数目为锁列表选择初始值，初始值位于你计算出的上限和下限之间。

maxlocks 配置参数的计算方法如下：

maxlocks = 100 * (512 锁/应用程序 * 128 字节/锁 * 2)/(locklist * 4096 字节)

上述公式允许任何应用程序持有的锁是平均数的两倍。如果只有几个应用程序并发地运行，那么可以增大 maxlocks，因为在这些条件下锁列表空间中不会有太多争用。

锁升级会在以下两种情况下被触发：

- 由某个应用程序请求的锁占用的内存空间超出了 maxlocks 和 locklist 的乘积大小。这时，数据库管理器将试图通过为提出锁请求的应用程序申请表锁，并释放行锁来节省空间。
- 在数据库中已被加上的全部锁占用的内存空间超出了 locklist 定义的大小。这时，数据库管理器也将试图通过为提出锁请求的应用程序申请表锁，并释放行锁来节省空间。

虽然锁升级过程本身并不用花很多时间，但是锁定整个表(相对于锁定个别行)降低了并发性，而且数据库的整体性能可能会由于对受锁升级影响的表的后续访问而降低。

在设计良好的数据库中，很少发生锁升级。如果锁升级将并行性降低到不可接受的程度(由 *lock_escalation* 监视元素监视)，那么就需要分析问题并决定如何解决此问题。

锁升级是有可能失败的，比如，现在应用程序已经在表上加有 IX 锁，在表中的某些行上加有 X 锁，另一个应用程序又来请求表上的 IS 锁以及很多行上的 S 锁，这会由于申请的锁数目过多引起锁升级。数据库管理器试图为应用程序申请表上的 S 锁来减少需要的锁数目，但 S 锁与表上原有的 IX 锁冲突，锁升级不能成功。

如果锁升级失败，引起锁升级的应用程序将接到－912 的 SQLCODE。

下面我们举一个实际的例子。

首先，运行下面的命令以打开针对锁的 DB2 监视器：

```
db2 -v update monitor switches using lock on
db2 -v terminate
```

然后收集数据库快照：

```
db2 -v get snapshot for database on sample | grep -i lock
```

在快照输出中，检查下列各项：

```
Locks held currently = 21224
Lock waits = 24683
Time database waited on locks (ms) = 32875
Lock list memory in use (Bytes) = 87224
Deadlocks detected = 89
Lock escalations = 8
Exclusive lock escalations = 12
Agents currently waiting on locks = 0
Lock Timeouts = 0
Internal rollbacks due to deadlock = 0
```

如果“Lock list memory in use (Bytes)”超过定义的 LOCKLIST 大小的 50%，那么就增

加 LOCKLIST 数据库配置参数中 4KB 页的数量。

如果发生“Lock escalations>0”或“Exclusive lock escalations>0”，就应该增大 LOCKLIST 或 MAXLOCKS，抑或同时增大两者。查看“Locks held currently”、“Lock waits”、“Time database waited on locks (ms)”、“Agents currently waiting on locks”和“Deadlocks detected”中是否存在高值，如果有的话，就可能是差于最优访问计划、事务时间较长或应用程序并发问题的症状。

5.4.3 锁等待及调整案例

当应用程序不能够立刻得到为对象请求的锁时，应用程序将进入等待服务的队列，等待占用锁的应用程序提交或回滚来释放锁，这种情况称为锁等待。锁定超时检测是数据库管理器功能，用于防止应用程序在异常情况下无限时地等待释放锁。例如，事务可能正等待由另一个用户的应用程序挂起的锁，但那个用户已离开工作站，而没有允许应用程序落实释放该锁的事务。要避免在这种情况下应用程序继续等待，将 locktimeout 配置参数设置为任何应用程序应等到获取锁的最长时间。这可以帮助避免全局死锁的情况发生。如果锁请求处于暂挂的时间大于 locktimeout 值，那么请求应用程序将接收到错误并将其事务回滚。locktimeout 的默认值为－1，关闭锁定超时检测；如果出现锁等待，应用程序将会出现无穷等待现象。例如，如果 APPL1 尝试获取已由 APPL2 挂起的锁，那么 APPL1 在超时周期到期时返回 SQLCODE－911(SQLSTATE 40001)，原因码为 68。对于生产系统中的 OLAP，locktimeout 一开始为 60(秒)比较好，对于 OLTP 大约为 10 秒比较好。对于开发环境，应该使用－1 以识别和解决锁等待的情况。如果有大量的并发用户，可能需要增加 locktimeout 时间以避免回滚。

如果快照监控结果输出中的“Lock Timeouts”是较高的数，那么可能是由以下原因造成的：

- locktimeout 的值太低
- 某个事务持有锁的时间有所延长
- 锁升级

锁等待可能会造成如下几种结果：

- 引起死锁(死锁是锁等待的特例)，由死锁检测器处理。
- 等待超时，等待的应用程序返回 SQLCODE –911 和原因代码 68，并自动回滚。超时的时间由数据库配置参数 locktimeout 设置，单位为秒。比如 locktimeout 的值为 60，如果进行锁等待的应用程序在 60 秒后还不能得到所需的锁，应用程序将自动回滚。
- 如果 locktimeout 的值被设置为－1，应用程序将永远等待，直到能够获得所需要的锁。

- 对于表、行、数据分区和 MDC 块锁定，应用程序可使用 SET CURRENT LOCK TIMEOUT 来覆盖数据库级别的 locktimeout 设置。

有时候，锁等待情形会导致锁超时，而锁超时又会导致事务被回滚。锁等待导致锁超时所需的时间段由数据库配置参数 locktimeout 指定。锁超时分析最大的问题是，不知道下一次的锁超时何时发生。为了捕捉死锁，可以创建死锁事件监视器。每当出现死锁时，这个死锁事件监视器便写一个条目。同样，对于锁超时也有类似的事件监视器。下面的语句可以创建锁相关的事件监控器。

```
    db2 "create event monitor LOCKEVMON for locking write to unformatted event
table(table LOCKEVMON in db2tool PCTDEACTIVATE 85) autostart"
```

分析锁事件监视器所生成的数据，需要使用基于 Java 的通用 XML 解析器工具 db2evmonfmt，可以将生成可读的纯文本输出(文本版本)或格式化 XML 输出。根据指定的参数，db2evmonfmt 工具将确定事件监视器数据的解析方式以及所要创建的输出类型。

db2evmonfmt 工具以 Java 源代码的形式提供。在使用此工具之前，必须通过执行下列步骤来设置和编译此工具：

(1) 在 sqllib/samples/java/jdbc 目录中找到源代码

(2) 按照 Java 源文件中嵌入的指示信息来设置和编译此工具

还可以随意修改源代码以更改输出。

此工具使用 XSLT 样式表将事件数据变换为带格式文本。不需要理解这些样式表。此工具将根据事件监视器类型自动装入正确的样式表并变换事件数据。每个事件监视器都将在 sqllib/samples/xml/data 目录中提供默认样式表。此工具还提供了下列过滤选项：

- 事件标识
- 事件时间戳记
- 事件类型
- 工作负载名
- 服务类名
- 应用程序名

下面的命令是一个简单的用法示例：

```
    cd /tmp
    cp ~/sqllib/samples/java/jdbc/db2evmonfmt.java
~/sqllib/samples/java/jdbc/DB2EvmonLocking.xsl
    ~/sqllib/java/jdk64/bin/javac db2evmonfmt.java
```

```
    ~/sqllib/java/jdk64/bin/java db2evmonfmt -d DBNAME -ue LOCKEVMON -ftext
> db2locks.out
```

一旦定位引起锁等待的 SQL 语句，如果该 SQL 语句写的效率很低下，可以考虑对该 SQL 语句做出调整；如果该 SQL 语句中没有创建最合理的索引，可以考虑用 db2advis 工具为该 SQL 语句创建最合理的索引。

传统的锁定方法会导致应用程序互相阻塞。当某个应用程序必须等待另一个应用程序释放锁定时，阻塞就会发生。用于处理这种阻塞的影响的策略通常会提供一种机制以指定最大可接受的阻塞持续时间，这就是应用程序在不能获取锁定的情况下在返回之前等待的时间。

以前，只能在数据库级别通过更改 locktimeout 数据库配置参数的值来指定时间。现在，锁等待方式策略通过新的 SET CURRENT LOCK TIMEOUT 语句指定，此语句更改 CURRENT LOCK TIMEOUT 专用寄存器的值。CURRENT LOCK TIMEOUT 专用寄存器指定在不能获取锁定错误之前等待锁定的秒数。

5.4.4　死锁及调整案例

死锁的产生是由于锁请求双方都彼此持有对方所需要的锁，这种情况下又去请求锁而导致死锁的产生。

下面我们通过一个典型的例子来说明死锁的概念。

假定事务 1 在表 A 上获取了互斥(X)锁，而事务 2 在表 B 上获取了互斥(X)锁。现在，假定事务 1 尝试在表 B 上获取互斥(X)锁，而事务 2 尝试在表 A 上获取互斥(X)锁。这两个事务的处理都将被挂起，直到同意第二个锁请求为止。但是，因为在任何一个事务释放自身目前持有的锁(通过执行或回滚操作)之前，这两个事务的锁请求都不会被同意，而且因为这两个事务都不能释放各自目前持有的锁(因为它们都已挂起并等待锁)，所以它们都陷入了死锁循环。图 5-11 说明了这个死锁场景。

当死锁循环发生时，除非某些外部代理进行干涉，否则涉及的所有事务将无限期地等待释放锁。在 DB2 UDB 中，用于处理死锁的代理是被称为死锁检测器的异步系统后台进程。死锁检测器的唯一职责是定位和解决在锁定子系统中找到的任何死锁。每个数据库都有自己的死锁检测器，在数据库初始化过程中激活。激活之后，死锁检测器在大多数时间处于“休眠”状态，但会以预置的时间间隔被“唤醒”，以确定锁定子系统中是否存在死锁循环。如果死锁检测器在锁定子系统中发现死锁，那么随机选择死锁涉及的事务，终止并回滚。选择的事务收到 SQL 错误编码(－911)，该事务获得的所有锁都被释放。这样，剩下的事务就可以继续执行了，因为死锁循环已经被打破了。

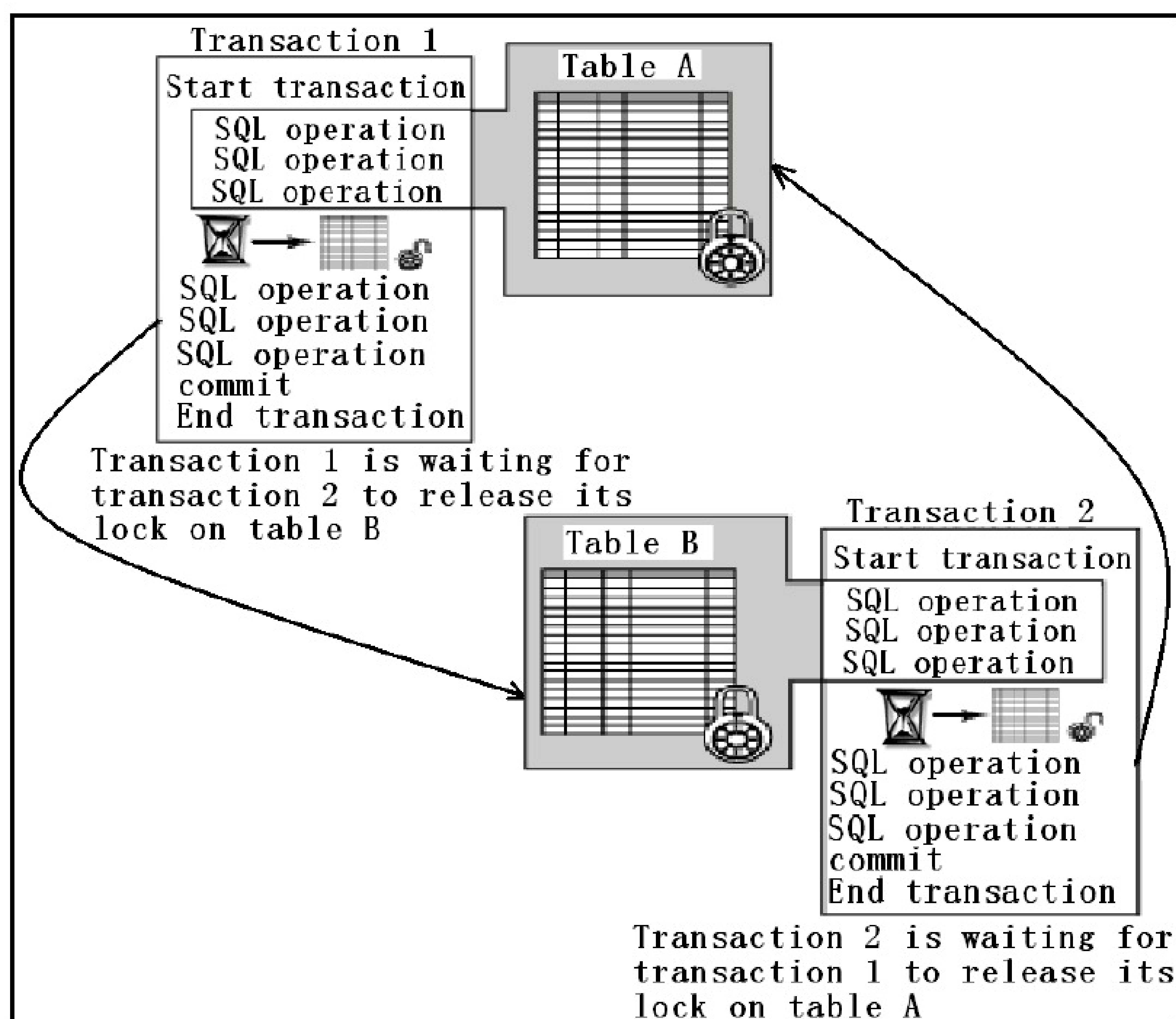


图 5-11　死锁循环示例

这就是死锁的典型情况，两个应用程序互相持有对方所需要的锁，在得不到自己所需要的锁的情况下，也不会释放现有的锁。

下面我们再举一个死锁的例子。

模拟一个死锁场景。为了方便模拟，我们将当前提交关闭：

```
$db2 update db cfg for test using cur_commit off
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
$db2stop force
SQL1064N  DB2STOP processing was successful.
$db2start
SQL1063N  DB2START processing was successful.
```

打开两个单独的命令行 CLP 窗口。

(1) 在第一个窗口中(图 5-12 所示)，执行以下命令(在每个命令后面按回车键)：

```
UPDATE COMMAND OPTIONS USING c OFF
(注意：这会关闭自动提交)
CONNECT TO TEST
```

```
CREATE TABLE deadtable (c1 INTEGER)
COMMIT
INSERT INTO deadtable VALUES (1)
```

```
db2 => UPDATE COMMAND OPTIONS USING c OFF
DB20000I  The UPDATE COMMAND OPTIONS command completed successfully.
db2 => CONNECT TO TEST

   Database Connection Information

 Database server        = DB2/AIX64 9.7.6
 SQL authorization ID   = DB2INST4
 Local database alias   = TEST

db2 => CREATE TABLE deadtable (c1 INTEGER)
DB20000I  The SQL command completed successfully.
db2 => COMMIT
DB20000I  The SQL command completed successfully.
db2 => INSERT INTO deadtable VALUES (1)
DB20000I  The SQL command completed successfully.
```

图 5-12 死锁场景 1

(2) 在第二个窗口中(图 5-13 所示)，执行以下命令：

```
UPDATE COMMAND OPTIONS USING c OFF
(注意：这会关闭自动提交)
CONNECT TO TEST
INSERT INTO deadtable VALUES(2)
SELECT * FROM deadtable
(你会看到光标不动了)
```

```
db2 => UPDATE COMMAND OPTIONS USING c OFF
DB20000I  The UPDATE COMMAND OPTIONS command completed successfully.
db2 => CONNECT TO TEST
SQL0752N  Connecting to a database is not permitted within a logical unit of
work when the CONNECT type 1 setting is in use.  SQLSTATE=0A001
db2 => INSERT INTO deadtable VALUES(2)
DB20000I  The SQL command completed successfully.
db2 => SELECT * FROM deadtable
█
```

图 5-13 死锁场景 2

(3) 在完成第(2)步之后，等待大约 15 秒。然后，在第一个窗口中，执行以下命令(图 5-14 所示)：

```
SELECT * FROM deadtable
```

```
db2 => INSERT INTO deadtable VALUES (1)
DB20000I  The SQL command completed successfully.
db2 => SELECT * FROM deadtable

C1
-----------
          1

  1 record(s) selected.

db2 =>
```

图 5-14　在第一个窗口中执行命令

(4) 这时会发生死锁，因为第一个窗口和第二个窗口都在等待对方释放锁，必须让其中某个事务回滚，才能打破锁冲突。DB2 死锁监视器进程会在 10 秒内选择某个事务并回滚。10 秒是默认设置，也可以通过数据库配置参数 DLCHKTIME 来设置。一个窗口将返回查询结果，另一个窗口将返回死锁消息(图 5-15 所示)。

```
db2 => UPDATE COMMAND OPTIONS USING c OFF
DB20000I  The UPDATE COMMAND OPTIONS command completed successfully.
db2 => CONNECT TO TEST
SQL0752N  Connecting to a database is not permitted within a logical unit of
work when the CONNECT type 1 setting is in use.  SQLSTATE=0A001
db2 => INSERT INTO deadtable VALUES(2)
DB20000I  The SQL command completed successfully.
db2 => SELECT * FROM deadtable
SQL0911N  The current transaction has been rolled back because of a deadlock
or timeout.  Reason code "2".  SQLSTATE=40001
db2 =>
```

图 5-15　第二窗口返回死锁消息

DB2 利用名为死锁检测器的后台进程进行死锁检测，该进程每隔一定的时间进行一次检测，一旦发现死锁，该进程就会选择某个牺牲者。牺牲者将自动回滚，释放掉占用的锁并返回 SQLCODE –911 和原因代码 2，死锁可以消除。

注意，DB2 死锁检测器会随机选择牺牲者，因此，不能确定哪个更新将发生回滚。在上面描述的场景中，第二个更新发生回滚，但是你在测试时，也许是第一个更新发生回滚。

dlchktime 是设置死锁检查间隔的配置参数，该参数以毫秒为单位，有效范围是 1000ms 至 600000ms。该值过高将增加应用程序等待死锁被发现的时间，如果过低，死锁检测的间隔虽然加快，但却降低了少许运行性能。该参数的默认值为 10000ms(10 秒)。

可能引起死锁的情况及影响程度包括：

- 应用程序的隔离级别采用可重复读和读稳定性(主要)
- 锁升级(主要)
- 锁转换(次要)
- 编目表的更改(中等)

- 参照完整性的约束(次要)

确定并查看快照输出结果中“Deadlocks detected”中是否存在高值，如果有的话，就可能是差于最优访问计划、事务时间较长或应用程序并发问题的症状。如果要发现死锁，那么需要创建针对死锁的事件监视器或 db2pd 工具。事件监视器带有详细信息，以便查看当前正在发生的事情。

```
create event monitor deadlock for deadlocks write to table
```

当我们在事件监视结果中定位了引起死锁的 SQL 语句后，如果该 SQL 语句写的效率很低，可以考虑对该 SQL 语句进行调整；如果该 SQL 语句中没有创建最合理的索引，可以考虑用 db2advis 工具为该 SQL 语句创建最合理的索引。

最小化死锁建议

- 在整个应用程序中，总是按相同次序访问资源可以最小化死锁。例如，如果某个应用程序组件将要访问表 A，然后是表 B，接着是表 C；而另一个应用程序组件需要访问表 A 和 C，那么第 2 个组件应该遵循先 A 后 C 的访问次序。
- 对于 DB2 V9 以前的版本，导致死锁的常见原因是锁列表数据库配置参数的大小不足，尤其是使用默认值时，因此应该合理地设置 LOCKLIST 和 MAXLOCKS 配置参数。默认情况下 DB2 V9 使用 STMM，它会调整锁列表大小以避免可能由此引起的锁升级和死锁。
- 确保参照完整性(Referential Integrity，RI)关系中的依赖表拥有与外键匹配的索引。

5.5　锁相关的性能问题总结

调整锁定以实现并行性和数据完整性时，应考虑下列准则：

- 频繁使用 COMMIT 语句来创建较小的工作单元以使许多用户可以并发访问数据。当应用程序在逻辑上一致时，即当更改的数据一致时，发出 COMMIT 语句。当发出 COMMIT 时，释放锁定，与声明了 WITH HOLD 的游标相关的表锁定除外。
- 在发出 COMMIT 语句之前，应先关闭 CURSOR WITH HOLD。在某些情况下，在结果集关闭并且事务落实后锁定仍然存在。在发出 COMMIT 语句之前关闭 CURSOR WITH HOLD 可确保释放锁定。
- 指定适当的隔离级别。即使应用程序很少读取行，也会获得锁定，所以落实只读工作单元仍很重要。这是因为在只读应用程序中通过可重复读、读稳定性及游标稳定性隔离级别可以获得共享锁定。借助于可重复读和读稳定性，可挂起所有锁

定，直到发出 COMMIT，这可以阻止其他进程更新锁定的数据，除非使用 WITH RELEASE 子句关闭游标。此外，甚至在使用动态 SQL 语句的未提交读应用程序中也要获得目录锁定。数据库管理器确保应用程序不检索未提交数据(其他应用程序已经更新但尚未落实的行)，除非正在使用未提交读隔离级别。

- 适当地使用 LOCK TABLE 语句。该语句锁定整个表，但只锁定 LOCK TABLE 语句中指定的表。不锁定指定表的父表和从属表。必须确定是否需要锁定其他可访问的表以便在并行性与性能方面达到期望的结果。在工作单元被落实或回滚之前不释放锁定。

以共享方式锁定表

想要访问在时间上一致的数据——表在特定时间点的最新数据。如果表活动频繁，那么确保整个表保持稳定的唯一方法是将其锁定。例如，应用程序想要抽取表的快照。但是，在应用程序需要处理表的一些行期间，其他应用程序正在更新还没有处理的行。可重复读允许这样的操作，但你不希望这样。

另一种方法是，应用程序可以发出 LOCK TABLE IN SHARE MODE 语句：无论是否检索到行，都不能更改任何行。然后可以按需要检索任意多行，应了解已经检索的行就在检索它们之前还没有被更改。

使用 LOCK TABLE IN SHARE MODE，其他用户可从表中检索数据，但不能更新、删除表中的行或将行插入表中。

以互斥方式锁定表

想要更新表的大部分。相对于更新表时锁定每一行，然后在落实所有更改时解锁行而言，该方式阻止所有其他用户访问该表的开销少并且更有效。

使用 LOCK TABLE IN EXCLUSIVE MODE，所有其他用户都被锁定在外；没有其他应用程序可以访问该表，除非它们是未提交读应用程序。

- 在应用程序中使用 ALTER TABLE 语句。

带 LOCKSIZE 参数的 ALTER TABLE 语句是 LOCK TABLE 语句的备用语句。LOCKSIZE 参数允许指定下一次表访问的 ROW 锁定或 TABLE 锁定的锁定详细程度。

当新建表时，选择 ROW 锁定同选择默认锁定大小无任何区别。选择 TABLE 锁定可提高查询性能，方法是限制需要获取的锁定的数目。但是，并行性可能由于所有锁定位于完整的表上而降低。仍然会对所有其他操作执行行级锁定，并对键插入行级锁定以保护可重复读(RR)扫描程序。

- 关闭游标以释放它们挂起的锁定。

当使用包括 WITH RELEASE 子句的 CLOSE CURSOR 语句关闭游标时，数据库管理

器尝试释放所有为游标挂起的读锁定。表读锁定是 IS、S 和 U 表锁定。行读锁定是 S、NS 和 U 行锁定。块读锁定是 IS、S 和 U 块锁定。

WITH RELEASE 子句对正在 CS 或 UR 隔离级别下操作的游标不起作用。当对正在 RS 或 RR 隔离级别下操作的游标指定 WITH RELEASE 子句后，将结束那些隔离级别的一些保证。明确地说，RS 游标可能经历不可重复读现象，而 RR 游标不会发生不可重复读或幻像读现象。

如果原来是 RR 或 RS 的游标通过使用 WITH RELEASE 子句关闭后重新打开，那么将获得新的读锁定。

在 CLI 应用程序中，DB2 CLI 连接属性 SQL_ATTR_CLOSE_BEHAVIOR 可用来获得与 CLOSE CURSOR WITH RELEASE 相同的结果。

5.6　锁与应用程序设计

为了确定锁定属性，可以将应用程序处理划分为下列类型的其中一种：

- 只读：此类型包括所有这样的选择语句，它们本身是只读的，并具有显式的 FOR READ ONLY 子句；或者是模糊的(但是查询编译器因为 PREP 或 BIND 命令指定了 BLOCKING 选项的值而假定它们是只读的)。此处理类型只要求“共享”锁定(S、NS 或 IS)。
- 更改意向：此类型包括具有 FOR UPDATE 的 SELECT 语句，或者查询编译器解释有歧义的语句以表示想进行更改。此类型使用“共享”和“更新”锁定(对于行，为 S、U 和 X；对于表，为 IX、U 和 X)。
- 更改：此类型包括 UPDATE、INSERT 及 DELETE，但不包括 UPDATE WHERE CURRENT OF 或 DELETE WHERE CURRENT OF。此类型要求“互斥”锁定(X 或 IX)。
- 受控游标：此类型包括 UPDATE WHERE CURRENT OF 和 DELETE WHERE CURRENT OF。此类型也要求“互斥”锁定(X 或 IX)。

根据子查询语句的结果，在目标表中进行插入、更新或删除数据的语句，执行两种类型的处理：只读处理规则确定对在子选择语句中返回的表的锁定；更改处理规则确定对目标表的锁定。

优化器假定应用程序必须检索由 SELECT 语句指定的所有行，此假设最适合于 OLTP 和批处理环境。但是，在“浏览”应用程序中，查询经常定义可能很大的答案集，但它们只检索前几行，通常只检索填满屏幕所需的那么多行。

要提高这种应用程序的性能，可以按下列方式修改 SELECT 语句：

- 使用 FOR UPDATE 子句指定可由后续定位的 UPDATE 语句更新的列。
- 使用 FOR READ/FETCH ONLY 子句使返回的列为只读的。
- 使用 OPTIMIZE FOR *n* ROWS 子句授予检索整个结果集中前 *n* 行的优先级。
- 使用 FETCH FIRST *n* ROWS ONLY 子句仅检索指定的几行。
- 使用 DECLARE CURSOR WITH HOLD 语句每次检索一行。

注意：

如果使用 FOR UPDATE、FETCH FIRST n ROWS ONLY 或 OPTIMIZE FOR n ROWS 子句，或者声明游标为“滚动”，那么会影响行分块。

FOR UPDATE 子句

FOR UPDATE 子句通过仅包含可由后续定位的 UPDATE 语句更新的列来限制结果集。如果不带列名指定 FOR UPDATE 子句，那么包括表或视图的全部可更新列。如果指定列名，那么每个名称必须是非限定的，并且必须标识表或视图的某一列。

在下列情况下，不能使用 FOR UPDATE 子句：

- 不能删除与 SELECT 语句关联的游标。
- 选择的列中至少有一列是目录表的不可更新列，并且没有在 FOR UPDATE 子句中排除掉。

由于相同的目的，可将 DB2 CLI 连接属性 SQL_ATTR_ACCESS_MODE 用于 CLI 应用程序中。

FOR READ ONKY 或 FOR FETCH ONLY 子句

FOR READ ONLY 或 FOR FETCH ONLY 子句确保返回只读结果。因为在被定义为只读的视图中，SELECT 的结果表也是只读的，所以允许此子句但没有作用。

对于允许更新和删除的结果表，如果数据库管理器可以检索数据块而不是互斥锁定，那么指定 FOR READ ONLY 可能会提高 FETCH 操作的性能。不要对用于定位的 UPDATE 或 DELETE 语句的查询使用 FOR READ ONLY 子句。

由于相同的目的，可将 DB2 CLI 连接属性 SQL_ATTR_ACCESS_MODE 用于 CLI 应用程序。

OPTIMIZE FOR *n* ROWS 子句

OPTIMIZE FOR 子句声明只想检索结果的某个子集或优先检索前几行。优化器然后可以优先选择把检索前几行的响应时间减至最短的访问方案。另外，作为单个块发送到客户机的行数由 OPTIMIZE FOR 子句中的“n”值来限制。因此，OPTIMIZE FOR 子句既影响服务器从数据库检索合格行的方式，又影响将合格行返回到客户机的方式。

例如，假设定期查询职员表来查找具有最高工资的职员：

```
SELECT LASTNAME,FIRSTNAME,EMPNO,SALARY
        FROM EMPLOYEE        ORDER BY SALARY DESC
```

上面定义了基于 SALARY 列的降序索引。但是，因为职员是按职员号排序的，所以工资索引可能很难集群。为了尽量避免许多随机的同步 I/O，优化器将可能选择使用列表预取访问方法，此方法需要对合格的所有行的行标识排序。在将前几个合格行返回至应用程序前，此排序可能导致延迟。要防止此延迟，可向语句添加 OPTIMIZE FOR 子句，如下所示：

```
SELECT LASTNAME,FIRSTNAME,EMPNO,SALARY FROM EMPLOYEE
          ORDER BY SALARY DESC        OPTIMIZE FOR 20 ROWS
```

在此情况下，优化器可能选择直接使用 SALARY 索引，因为只检索具有最高工资的 20 个职员。不管可以将多少行分块，而只把由每 20 行组成的块返回至客户机。

使用 OPTIMIZE FOR 子句，优化器优先选择可以避免大量操作或中断行流动(如排序)的访问方案。使用 OPTIMIZE FOR 1 ROW 最有可能影响访问路径。使用此子句可以有下列作用：

- 连接方法可以更改。嵌套循环连接是非常可能的选择，因为具有低开销成本，并且通常在检索少量行时更有效率。
- 与 ORDER BY 子句匹配的索引更可能，因为对于 ORDER BY 不需要排序。
- 列表预取不太可能，因为此访问方法需要排序。
- 顺序预取不太可能，因为只需要知道少量的几行。
- 在连接查询中，在 ORDER BY 子句中包含列的表可能被选作外部表，前提是外部表上的某个索引提供 ORDER BY 子句所需的排序。

OPTIMIZE FOR 子句虽然适用于所有优化级别，但却在优化级别 3 和更高级别下工作得最好。因为级别 3 以下的级别使用“贪婪”连接枚举方法，此方法有时会产生不能使它们自己很快检索前几行的多表连接的访问方案。

OPTIMIZE FOR 子句不阻止检索全部合格行。如果确实要检索全部合格行，那么总耗用时间可能大大高于优化器为整个答案集进行优化所需的时间。

如果已打包的应用程序使用调用级接口(DB2 CLI 或 ODBC)，可在 db2cli.ini 配置文件中使用 OPTIMIZE FOR N ROWS 关键字，让 DB2 CLI 自动将 OPTIMIZE FOR n ROWS 子句追加至每个查询语句的末尾。

FETCH FIRST *n* ROWS ONLY 子句

FETCH FIRST *n* ROWS ONLY 子句设置可检索的最大行数。将结果表限制为只包含前几行可提高性能。无论结果集可能另外包含多少行，只检索 *n* 行。

如果同时指定 FETCH FIRST 子句和 OPTIMIZE FOR 子句，那么这两个值中较小的那个影响通信缓冲区(RQRIOBLK，最大请求 I/O 块)大小。为了达到最优化，将这两个值看作互不相关的。

DECLARE CURSOR WITH HOLD 语句

当用包括 WITH HOLD 子句的 DECLARE CURSOR 语句声明游标时，在落实该事务时任何打开的游标仍然打开，并且释放所有锁定(保护打开的 WITH HOLD 游标的当前游标位置的锁定除外)。

如果回滚事务，那么关闭所有打开的游标并释放所有锁定。

5.7 锁监控工具

在 DB2 中对锁进行监控有很多工具：快照监控、事件监控和 db2pd。下面分别举例说明。

1. 快照监控方式

在使用快照方式对锁进行监控之前，必须把监控锁的开关打开，可以从实例级别或会话级别打开，具体命令如下：

```
db2 update dbm cfg using dft mon lock on --实例级别
db2 update monitor switches using lock on  --会话级别，推荐使用
```

开关打开后，可以执行下列命令来进行锁监控：

```
db2 get snapshot for locks on dev
```

数据库 dev 中具体锁的详细信息如下：

```
                 Database Lock Snapshot
Database name                          = DEV
Database path                          = /db2/DEV/db2dev/NODE0000/SQL00001/
Input database alias                    = DEV
Locks held                              = 49
Applications currently connected            = 38
Agents currently waiting on locks           = 6
```

```
Snapshot timestamp                                 = 08-15-2013 15:26:00.951134
Application handle                                 = 6
Application ID                                    = *LOCAL.db2dev.130815021007
Sequence number                                   = 0001
Application name                                  = disp+work
Authorization ID                                  = SAPR3
Application status                                 = UOW Waiting
Status change time                                 =
Application code page                               = 819
Locks held                                       = 0
Total wait time (ms)                                = 0
Application handle                                 = 97
Application ID                                    = *LOCAL.db2dev.130815060819
Sequence number                                   = 0001
Application name                                  = tp
Authorization ID                                  = SAPR3
Application status                                 = Lock-wait
Status change time                                 = 08-15-2013 15:08:20.302352
Application code page                               = 819
Locks held                                       = 6
Total wait time (ms)                                = 1060648
 Subsection waiting for lock                        = 0
  ID of agent holding lock                          = 100
  Application ID holding lock                        = *LOCAL.db2dev.130815061638
  Node lock wait occurred on                         = 0
  Lock object type                                = Row
  Lock mode                                     = Exclusive Lock (X)
  Lock mode requested                               = Exclusive Lock (X)
  Name of tablespace holding lock                      = PSAPBTABD
  Schema of table holding lock                        = SAPR3
  Name of table holding lock                         = TPLOGNAMES
  Lock wait start timestamp                          = 08-15-2013 15:08:20.302356
  Lock is a result of escalation                      = NO
List Of Locks
  Lock Object Name                                 = 29204
 Node number lock is held at = 0
 Object Type                                       = Table
 Tablespace Name                                   = PSAPBTABD
 Table Schema                                      = SAPR3
 Table Name                                        = TPLOGNAMES
 Mode                                              = IX
 Status                                            = Granted
 Lock Escalation                                   = NO
```

```
    Lock Escalation                              = NO
```

2. 事件监控方式

当使用事件监控器进行死锁监控时，具体步骤如下：

创建事件：

```
    db2 create event monitor dlock for deadlocks with details write to file
'$HOME/dir'
    db2 set event monitor dlock state 1
```

查看具体输出：

```
    db2evmon -db sample -evm dlock
    Deadlocked Connection ...
      Deadlock ID:   4
      Participant no.: 1
      Participant no. holding the lock: 2
      Appl Id: G9B58B1E.D4EA.08D387230817
      Appl Seq number: 0336
      Appl Id of connection holding the lock: G9B58B1E.D573.079237231003
      Seq. no. of connection holding the lock: 0126
      Lock wait start time: 06/08/2015 08:10:34.219490
      Lock Name       : 0x000201350000030E0000000052
      Lock Attributes : 0x00000000
      Release Flags   : 0x40000000
      Lock Count      : 1
      Hold Count      : 0
      Current Mode    : NS  - Share (and Next Key Share)
      Deadlock detection time: 06/08/2015 08:10:39.828792
      Table of lock waited on      : ORDERS
      Schema of lock waited on     : DB2INST1
      Tablespace of lock waited on : USERSPACE1
      Type of lock: Row
      Mode of lock: NS  - Share (and Next Key Share)
      Mode application requested on lock: X   - Exclusive
      Node lock occured on: 0
      Lock object name: 782
      Application Handle: 298
      Deadlocked Statement:
        Type     : Dynamic
        Operation: Execute
        Section  : 34
        Creator  : NULLID
```

```
        Package  : SYSSN300
        Cursor   : SQL CURSN300C34
        Cursor was blocking: FALSE
        Text     : UPDATE ORDERS  SET TOTALTAX = ?, TOTALSHIPPING = ?, LOCKED = ?,
TOTALTAXSHIPPING = ?, STATUS = ?, FIELD2 = ?, TIMEPLACED = ?, FIELD3 = ?, CURRENCY
= ?, SEQUENCE = ?, TOTALADJUSTMENT = ?, ORMORDER = ?, SHIPASCOMPLETE = ?,
PROVIDERORDERNUM = ?, TOTALPRODUCT = ?, DESCRIPTION = ?, MEMBER ID = ?,
ORGENTITY ID = ?, FIELD1 = ?, STOREENT ID = ?, ORDCHNLTYP ID = ?, ADDRESS ID
= ?, LASTUPDATE = ?, COMMENTS = ?, NOTIFICATIONID = ? WHERE ORDERS ID = ?
      List of Locks:
          Lock Name                  : 0x000201350000030E0000000052
          Lock Attributes             : 0x00000000
          Release Flags              : 0x40000000
          Lock Count                 : 2
          Hold Count                 : 0
          Lock Object Name            : 782
          Object Type                : Row
          Tablespace Name             : USERSPACE1
          Table Schema               : DB2INST1
          Table Name                 : ORDERS
          Mode                      : X   - Exclusive
          Lock Name                  : 0x00020040000029B30000000052
          Lock Attributes             : 0x00000020
          Release Flags              : 0x40000000
          Lock Count                 : 1
          Hold Count                 : 0
          Lock Object Name            : 10675
          Object Type                : Row
          Tablespace Name             : USERSPACE1
          Table Schema               : DB2INST1
          Table Name                 : BKORDITEM
          Mode                      : X   - Exclusive(略去后面信息)
```

也可以使用专门的死锁事件监控器，如采用 5.4.3 节中的 locking 事件监控器，这种锁事件监控器不仅可以记录锁超时事件信息，同时也记录死锁的事件信息。

3. db2pd 监控锁

图 5-16 展示了用于锁监视的 db2pd 选项。



图 5-16　用于锁监视的 db2pd 选项

- TranHdl：用于指定事务句柄，以便只监视由特定事务持有的锁。
- showlocks：这个子选项将锁的名称扩展成有意义的解释。对于行锁，该选项显示以下信息：表空间 ID、表 ID、分区 ID、页和槽。通过使用编目视图 SYSCAT.TABLES 上的查询，可以很容易将表空间 ID 和表 ID 映射到相应的表名。

下面展示了检查锁等待的情形：

```
db2pd -db sample -locks wait showlocks
Database Partition 0 -- Database SAMPLE -- Active -- Up 3 days 08:33:05
Locks:
Address    TranHdl    Lockname          Type       Mode Sts Owner     Dur
0x050A0240 6      02000600050040010000000052 Row        ..X  W   2         1
0x050A0DB0 2      02000600050040010000000052 Row        ..X  G   2         1
HoldCount  Att  ReleaseFlg
0    0x00 0x40000000   TbspaceID 2  TableID 6  PartitionID 0 Page 320 Slot 5
0    0x00 0x40000000   TbspaceID 2  TableID 6  PartitionID 0 Page 320 Slot 5
```

5.8 最大化并发性

对于好的数据库性能来说，最大化并发性非常重要。下面列出了一些详细的建议。

5.8.1 选择合适的隔离级别

选择隔离级别可以为应用程序提供可接受的最佳并发性。有几种方式可用来指定隔离级别，比如对 SQL 语句(只应用于该语句)使用 CURRENT ISOLATION 专用寄存器(应用于连接)，对 JDBC 连接对象进行指定(应用于连接)等。

5.8.2 尽量避免锁等待、锁升级和死锁

通过监控锁升级的发生(通过 DB2 状态监控器、db2diag.log、Windows 事件浏览器或其他性能监控器)，确保锁列表和 MAXLOCK DB2 配置参数足够大。锁列表大小不足将会导致 DB2 尝试将大量行锁“升级”到单个表级别的锁。如果升级失败，就会导致死锁；如果升级成功，又会极大地影响到并发性。

5.8.3 设置合理的注册变量

可以通过使用 3 个 DB2 注册变量——DB2_EVALUNCOMMITTED、DB2_SKIPDELETED 和 DB2_SKIPINSERTED 来提高并发性。如果不设置这三个注册变量，当用户正在更改(update)、插入(insert)或删除(delete)一行时，DB2 会在这一行加上排它锁(eXclusive)，别的

用户不能读写，除非使用 UR 隔离级别。

其实，目前市场上除了 Oracle 外所有的数据库，包括 DB2、Informix、SQL Server 和 Sybase，对锁的控制都是这种方式。而 Oracle 因为有回滚段(rollback segment)，所以在 Oracle 数据库中对于 insert 一行，回滚段记录插入记录的 rowid；对于 update 操作，回滚段记录更新字段的旧值(before image)；对于 delete 操作，回滚段记录整行的数据。由于 Oracle 有了回滚段，因此可以实现多版本读。所以在用 Oracle 数据库开发时，很少关注锁的情况，因为大部分情况下你都是可以读的，只不过有的时候大不了读以前的“before image”罢了。所以很多使用 Oracle 开发的用户在转向 DB2 开发时，都特别郁闷。而 DB2 为了改善应用程序并发性，就陆续引入了这 3 个变量。这 3 个变量并不会改变锁的本质，只不过是了解它们的工作方式和机制，以使我们根据业务逻辑来合理地设置调整以提高应用程序的并发性。这 3 个变量在实现原理上和 DB2 V9.7 新加的参数 CUR_COMMIT 是完全不同的，并且在 CUR_COMMIT 为 ON 的时候，会覆盖该参数包含的大部分场景，因此在下面的例子里，我们将 CUR_COMMIT 修改为 DISABLED。

下面我们先通过一个例子来说明没有这 3 个变量之前的一些锁的情况。假设 t1 表中有 5 条记录，分别为 11、22、33、44、55。其中第 2 条记录 22 被删除了，现在 session1 要重新插入一条新的记录 22；同时 session 2 执行了 db2 select * from t1 where id >11 and id<44，正常的话应该检索到 33 这条记录，但是由于现在插入的记录 22 也包含在这个谓词的限定范围内，这个时候 session 2 处于锁等待状态。

```
Session 1
db2 CONNECT TO SAMPLE
db2 +c "INSERT INTO t1 VALUES(22) "
```

```
Session 2
db2 CONNECT TO SAMPLE
db2 "SELECT * FROM t1 WHERE id >11
and id <44"
```

我们通过监控看到的效果如图 5-17 所示。

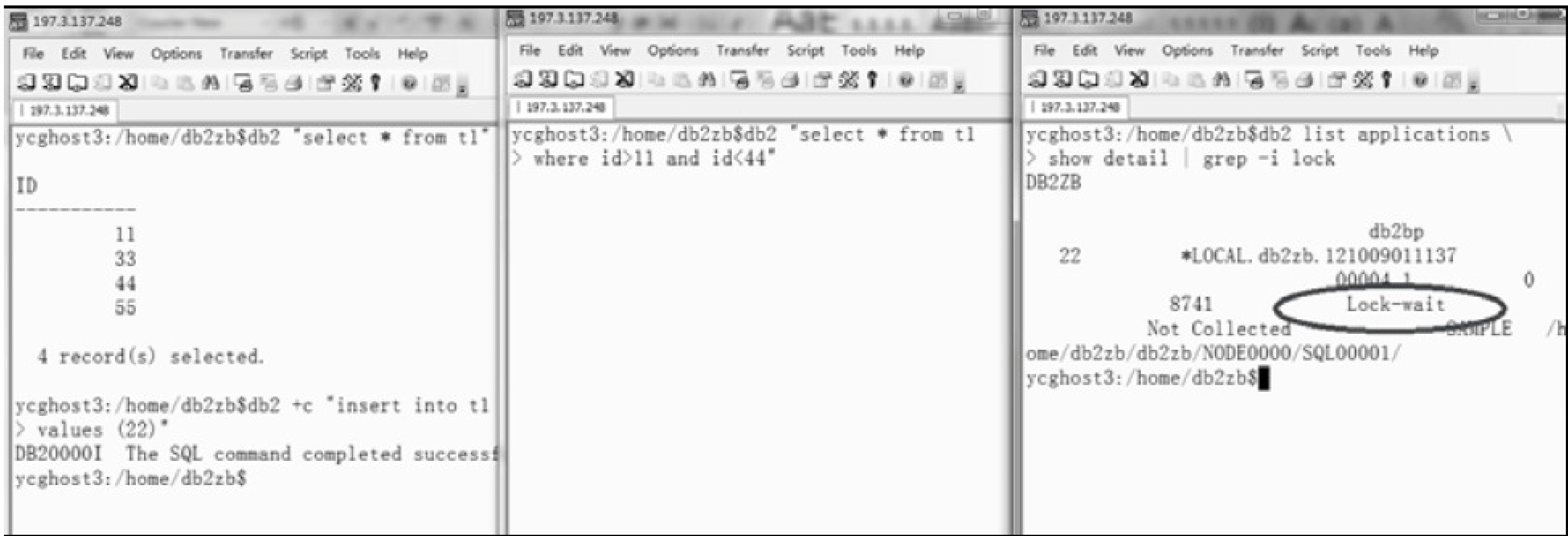


图 5-17　监控结果

从 DB2 的角度来说这好像是合理的，但是从用户和业务逻辑的角度来说，希望这个时候能够读取到数据，那么怎么解决这个矛盾呢？下面我们来仔细讲解这 3 个变量。

DB2_EVALUNCOMMITTED

当启用了 DB2_EVALUNCOMMITTED 这个 DB2 注册变量(=TRUE | ON | YES | 1)时，它将修改 DB2 中只读查询的行为，使之允许在索引扫描(必须是 type-2 索引，对于 type-1 索引该特性不受支持)或表访问时推迟锁，直到限定语句的所有谓词都是已知的。这个注册变量是为了可选地提高一些应用程序的并发性，其实质是允许读扫描推迟或避免行锁，直到适合特定查询的某条数据记录成为已知。

如果没有设置这个注册变量，DB2 将执行保守式的锁：在验证行是否满足查询的排除谓词之前，将锁定每个被访问的行。不管数据行是否被提交，以及根据语句的谓词是否被排除，对于索引扫描和表访问都执行这样的锁定操作。下面我们举一个简单的例子：

```
db2 create table t1(id int)
db2 insert into t1 values(11)
db2 commit
```

现在有两个会话分别发出了下面的 SQL 语句：

```
Session 1
db2 CONNECT TO SAMPLE
db2 +c "INSERT INTO t1 VALUES(22)"
```

```
Session 2
db2 CONNECT TO SAMPLE
db2 "SELECT * FROM t1 WHERE id = 11"
```

查看 Session 2 的状态，如图 5-18 所示。

第一条语句 DB2 +C "INSERT INTO TABLE T1 VALUES (22) "阻塞所有其他的扫描器，因为它持有行上的锁；如果第二个会话执行 DB2 "SELECT * FROM T1"，那么将被阻塞，直到事务 1 提交或回滚。但是假设第二条语句是 DB2 "SELECT * FROM T1 WHERE id=11"。在此情况下，即使事务 2 与列 ID=22 中的任何值(还没有被提交)都没有关系，也仍将被阻塞，处于锁等待状态。在 DB2 中，默认情况下将发生这一系列的事件，因为默认的隔离级别是 Cursor Stability(CS)。这种隔离级别表明，查询访问的任何一行在游标定位到该行时都必须被锁定。在语句 1 释放用于更新表 t1 的锁之前，语句 2 不能包含表 t1 第一行上的锁。如果 DB2 知道值 ID=11 不是语句 2 的数据请求的一部分(换句话说，在锁定行之前计算了谓词)，就可以避免阻塞，这是合情合理的，因为语句 2 不会尝试锁定表中的第一行。

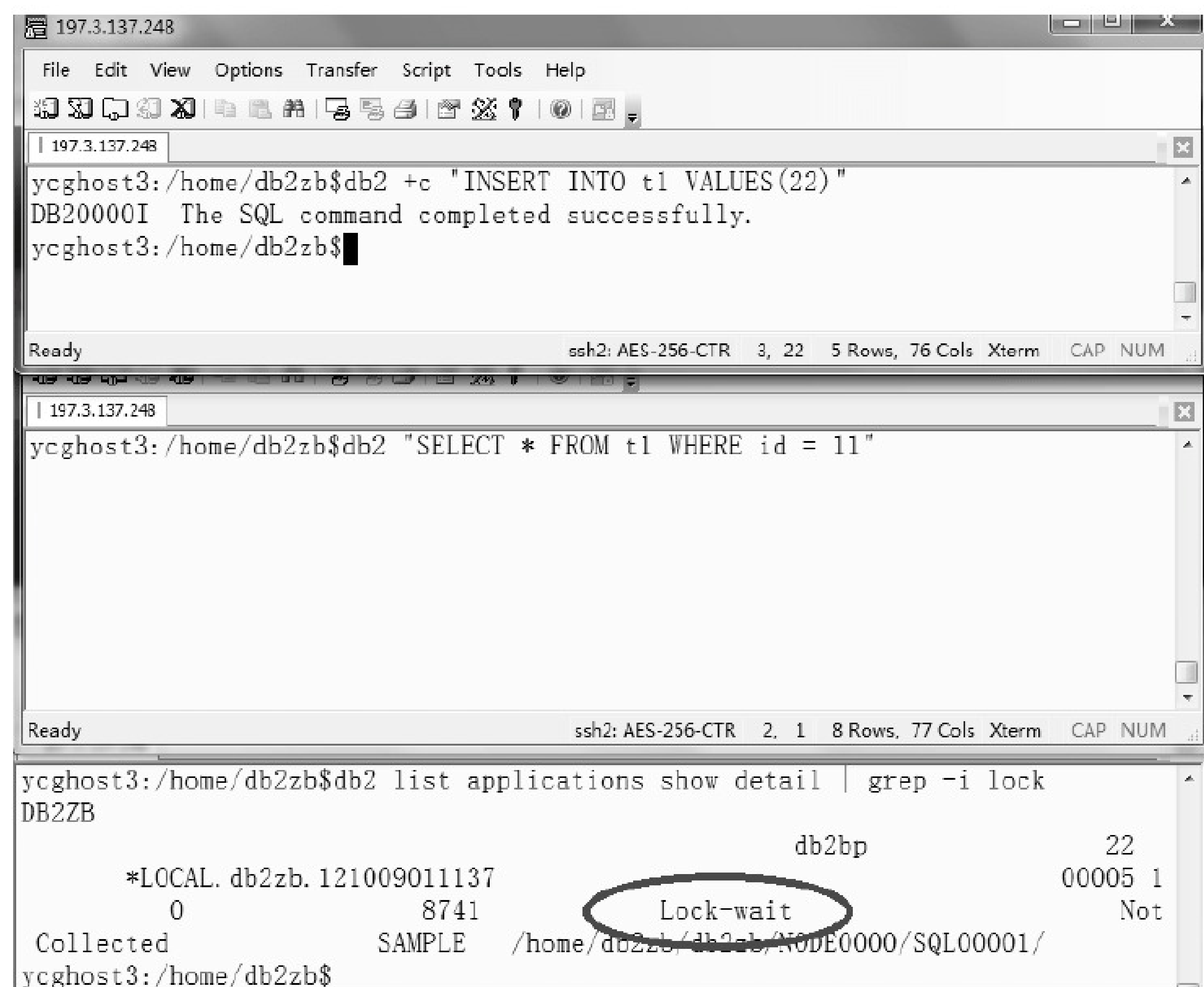


图 5-18 Session 2 的状态

现在启用 DB2_EVALUNCOMMITTED 注册变量，实例设置后需要重启：

```
db2set DB2_EVALUNCOMMITTED=ON
db2stop force
db2start
```

在启用实例后，重复刚才的实验，会发现第二条 SQL 语句"select * from t1 where id=11"可以执行而不会被阻塞。所以 DB2_EVALUNCOMMITTED 注册变量的作用是判断 SQL 谓词扫描的行是否有锁。如果没有，那么可以检索到数据。

DB2_EVALUNCOMMITTED 注册变量有以下限制：

- 必须启用 DB2_EVALUNCOMMITTED 注册表变量。
- 隔离级别必须是 CS 或 RS。
- 行级别锁定生效。
- 存在控制求值谓词。
- 未落实数据求值功能不适用于对系统目录表执行的扫描。
- 对于多维集群(MDC)或插入时间集群(ITC)表，可以对索引扫描延迟块级锁定；但是，不能对表扫描延迟块级锁定。
- 对于正在执行原位置表重组操作的表，不会发生锁定延迟。

- 对于索引扫描访存方案而言，不会将行级别锁定延迟到数据访问期间进行；而是，在移至表中的某行之前在索引访问期间锁定该行。
- 在表扫描期间将无条件地跳过已删除的行，但仅当 DB2_SKIPDELETED 注册表变量处于启用状态时，才会跳过已删除的索引键。

DB2_EVALUNCOMMITTED 变量影响 DB2 在游标稳定性(CS)和读稳定性(RS)隔离级别下的行锁机制。当启用该功能时，DB2 可以对未提交的插入(INSERT)或更新(UPDATE)数据进行谓词判断。如果未提交数据不符合该条语句的谓词判断条件，DB2 将不对未提交数据加锁，这样避免了因为要对未提交数据加锁而引起的锁等待状态，提高了应用程序访问的并发性。同时，DB2 在无条件进行表扫描时会忽略删除的行数据(不管是否提交)。

这里分两部分来看待，“对于插入(INSERT)或更新(UPDATE)而言，如果未提交数据不符合该条语句的谓词判断条件，那么 DB2 将不对未提交数据加锁”，这样虽然比不上 Oracle 对于符合该条语句的谓词判断条件可以从回滚段里面读出“before image”那样做到写不阻止读，但是起码在一定程度上缓解了锁的问题，不会因为插入(INSERT)或更新(UPDATE)一条记录造成整个表都锁住。这是个进步，也不会造成什么大的负面影响，

下面我们通过一个实验来说明这点：

```
db2 create table t1(id int)
db2 "insert into t1 values(11)"
db2 "insert into t1 values(22)"
db2 commit 现在表中有两条记录 11 和 22
```

现在两个会话发出了下面的 SQL 语句：

```
Session 1                              Session 2
db2 CONNECT TO SAMPLE                  CONNECT TO SAMPLE
db2 +c "delete from t1 where id=22"    db2 "SELECT * FROM t1"
```

在未设置 DB2_EVALUNCOMMITTED=ON 时，Session 2 肯定处于锁等待状态，在设置了 DB2_EVALUNCOMMITTED=ON 后，再来看看 Session 2 能否检索到数据：

```
Session 1 Session 2
                                     db2 CONNECT TO SAMPLE
                                     $db2 "select * from t1"
db2 CONNECT TO SAMPLE                ID
db2 +c "delete from t1 where id=22"  -----------
                                               11
                                     1 条记录已选择
```

通过上面的实验，我们发现在启用 DB2_EVALUNCOMMITTED=ON 时，对于删除操

作的处理，DB2 会无条件地在进行表扫描时忽略删除的行数据(不管是否提交)。个人觉得有很大的问题，通过上面的这个测试，一个事务删除一条记录并没有提交，另一个会话查询的时候已经没有这条记录了，这相当于 UR 隔离级别。这样显然是不符合业务要求的，与其这样还不如锁住。所以使用 DB2_EVALUNCOMMITTED=ON 时，删除的时候应该注意多多测试。

现在，在 t1 创建 type-2 索引，再来做刚才的那个实验：

```
db2 "create index index11 on t1(id)"
```

在两个命令行窗口(如图 5-19 所示)中分别发出下面的 SQL 语句：

```
Session 1                                       Session 2
  db2 CONNECT TO SAMPLE                         db2 CONNECT TO SAMPLE
  db2 create index index11 on t1(id)            db2 "select * from t1"
db2 +c "delete from t1 where id=22"             --锁等待挂起
```

我们在另一个窗口中查看 Session 2 的状态，发现 Session 2 处于锁等待状态，如图 5-19 所示。

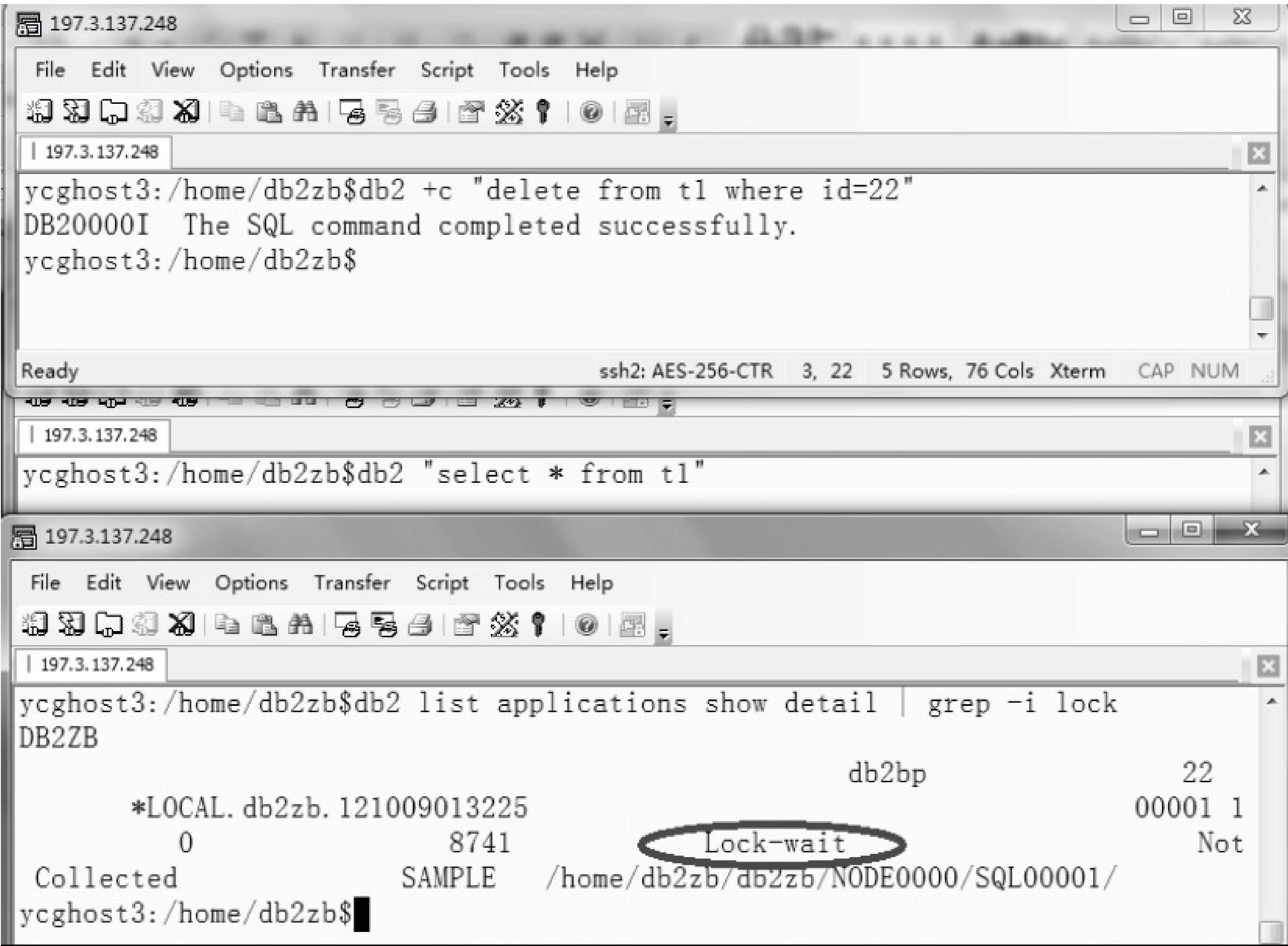


图 5-19 实验结果

当 DB2 环境中启用了 Evaluate Uncommitted 行为时，你应该清楚，谓词计算可能发生在未提交的数据上。而且在表扫描访问中，删除行被无条件忽略；而对于 type-2 索引扫描，被删除的键不会被忽略(除非你还设置了 DB2_SKIPDELETED 注册变量，DB2_SKIPDELETED 变量我们在稍后介绍)。如果要在环境中单独设置 DB2_SKIPDELETED 注册变量，DB2 将允许在表扫描访问时无条件地忽略被删除行，并忽略 type-2 索引扫描访问的伪删除索引键。

DB2_SKIPDELETED

变量 DB2_SKIPDELETED(被启用时=ON)将允许使用 Cursor Stability 或 Read Stability 隔离级别的语句，在索引扫描期间无条件地跳过被删除的键，而在表访问期间则无条件地跳过被删除的行。当 DB2_EVALUNCOMMITTED 被启用时，被删除的行会被自动跳过，但是除非同时启用了 DB2_SKIPDELETED，否则 type-2 索引中未提交的伪删除键不会被跳过。

在上面的实验中，我们发现当设置了 DB2_EVALUNCOMMITTED 变量时，如果表上有 type-2 索引，那么在读取数据时，被删除的索引键不会被忽略。这种情况下如果希望跳过被删除的键，可以通过设置 DB2_SKIPDELETED=ON 来实现，下面我们做个实验：

```
db2set DB2_SKIPDELETED=ON
db2stop force
db2start
```

设置生效后，我们接着做刚才的实验：

```
Session 1                                  Session 2
                                           db2 CONNECT TO SAMPLE
                                           db2 "select * from t1"
db2 CONNECT TO SAMPLE                      ID
db2 create index index11 on t1(id)         -----------
db2 +c "delete from t1 where id=22"              11

                                             1 条记录已选择
```

可以看到在设置 DB2_SKIPDELETED=ON 后，即使 t1 表上有 type-2 索引，在扫描的时候也仍然忽略这个删除的行。但是这在用的时候一定要结合业务逻辑使用，因为这种情况下等同于“脏读”，所以一定要多测试。

DB2_SKIPINSERTED

虽然当一行由于未提交的 INSERT 而被锁住的时候，这种行为是正确的；但是有些情况下应用程序的所有者希望 DB2 忽略正在等待提交的被插入的行，就好像它不存在一样(由于未提交 INSERT 的提交版本现在根本没有行，因此这是可能的)。例如，银行在下午 5 点左右想统计今天的业务量，这时只是想了解大概的业务量而不是精确的，这种情况下如果启用该变量，那么遗漏一两笔业务是可以接受的。

DB2_SKIPINSERTED=OFF 是默认设置，这使得 DB2 的行为和预期的一样：扫描器一直等到 INSERT 事务提交或回滚，然后返回数据——这和平常一样，取决于应用程序以及和业务逻辑相关的数据完整性的特征，这样可能合适，也可能不合适。例如，考虑涉及两个应用程序的业务流程——例如，信用评级应用程序和信用评分引擎，这两个应用程序使用相同的表来交换业务信息。应用程序 A 基于 Web 表单将数据插入数据库，应用程序 B 读这些数据。为了加快信用审批的速度，候选者通过信用评级应用程序表单转移，信息块通过表单中的'Steps'被发送到应用程序 B(通过公共的表)。当候选者完成信用评级应用程序流程中的每个步骤时，信息被发送。在这个环境中，数据必须由第二个应用程序按照表中给出的顺序来处理，以便当接下来要读的行要被应用程序 A 插入时，应用程序 B 必须等待，直到 INSERT 被提交。

如果设置 DB2_SKIPINSERTED=ON，DB2 将把未提交的 INSERT(只对于 CS 和 RS 隔离级别)看作它们还没有被插入。该特性提高了并发性，同时又不牺牲隔离语义。DB2 为扫描器实现了这种能力，通过锁属性和锁请求的反馈，使其忽略未提交的插入行，而不是等待。

下面我们举一个设置 DB2_SKIPINSERTED 变量前后的例子：

Session 1	Session 2
db2 CONNECT TO SAMPLE	db2 CONNECT TO SAMPLE
db2 "create index index11 on t1(id)"	db2 "select * from t1"
db2 +c "insert into t1 values(33)"	--挂起，处于锁等待状态

通过监控发现 Session 2 处于锁等待状态，如图 5-20 所示。

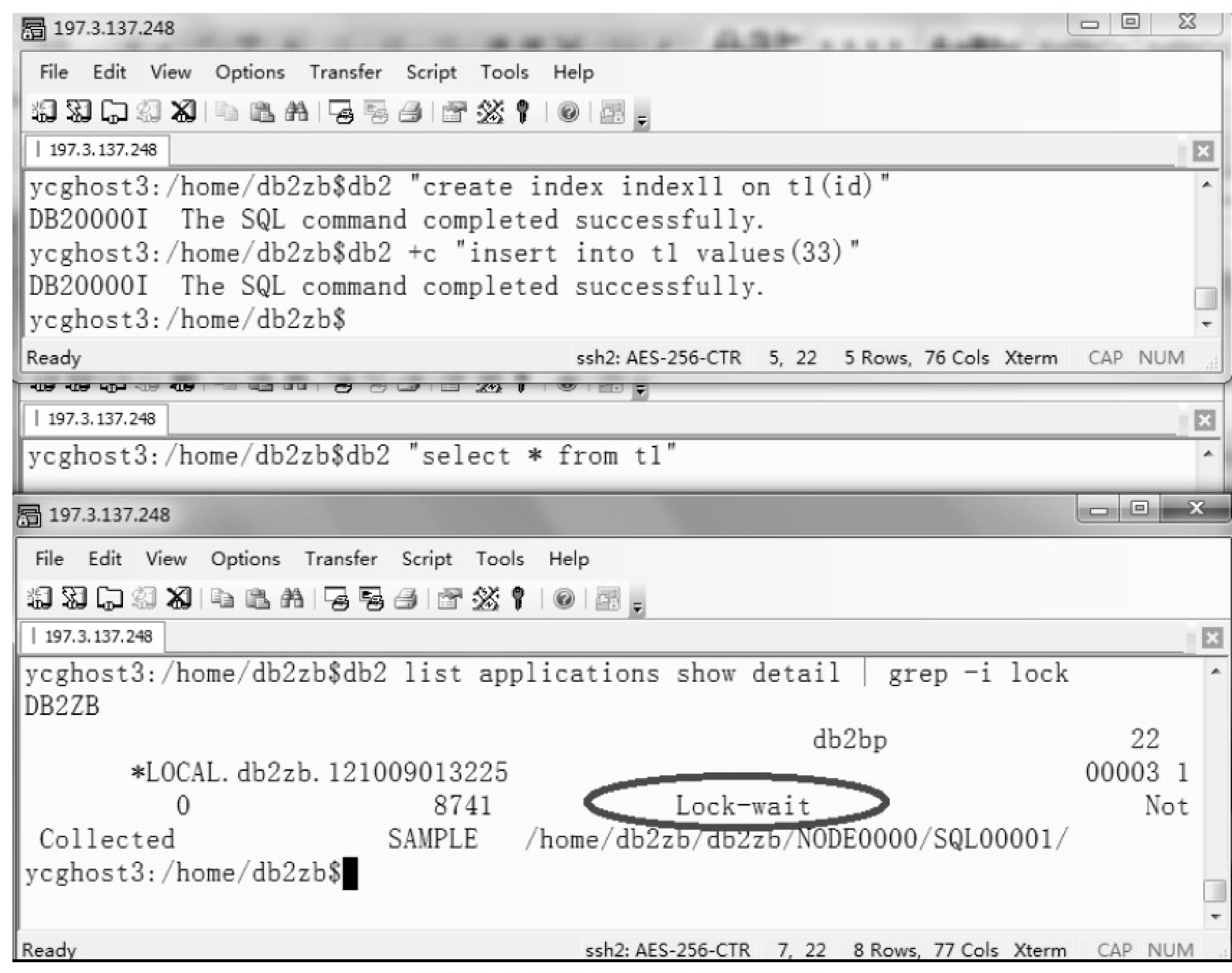


图 5-20 Session 2 的状态

如果在这种情况下 Session 2 希望能够跳过未提交 INSERT 的数据而得到数据，那么可以设置 DB2_SKIPINSERTED 注册变量：

```
db2set DB2_SKIPINSERTED=ON
db2stop force
db2start
```

然后重复刚才的实验，我们发现这个时候，Session 2 可以读到数据。

Session 1	Session 2
	db2 "select * from t1"
db2 CONNECT TO SAMPLE	ID
db2 "create index index11 on t1(id) "	-----------
db2 +c "insert into t1 values(33)"	11
	22
	2 条记录已选择。

5.9　本章小结

总的来说，这 3 个注册变量会影响到并发性。通过合理设置这些变量可以改善并发性，但是也会影响到应用程序的行为。建议在 DB2 开发设计的初期启用这些注册变量，从而在实现并发性增强后执行全面测试中的所有单元测试。

通过设置这 3 个注册变量，可以提高并发性，但是使用的时候一定要结合自己的业务逻辑来使用。根据个人经验，建议使用 DB2_EVALUNCOMMITTED=ON 和 DB2_SKIPINSERTED=ON。对于 DB2_SKIPDELETED 变量来说，使用的时候一定要充分地测试，因为这等同于使用 UR 隔离级别(虽然 DB2_SKIPINSERTED=ON 也等同于 UR，但是没有插入的数据反正也没有插入，读不到在业务上是可以接受的)。

总之，有了这 3 个变量，我们多了一种选择，这总比没有强。目前这 3 个变量都是实例级别的变量，如果能做成 SQL 级别或应用级别的就更好了。期待 DB2 能够在后续的版本中继续对并发做出改进。

第 6 章 索引设计与优化

索引是数据库性能优化的第一关，本身的特点使得它深受大家的喜爱——疗效好，见效快。无论是高手还是菜鸟，一般都会深深受益于索引。一个合适的索引，能使数据库性能提升几倍，甚至几十倍、几百倍也不足为奇。

本章主要讲解如下内容：

- 索引概念
- 索引的结构
- 索引访问机制
- 索引设计
- 索引创建原则与示例
- 影响索引性能的相关配置
- 索引维护
- 使用 db2advis 调整索引
- 索引调整总结

6.1 索引概念

6.1.1 索引优点

索引是表的一个或多个列的键值的有序列表。创建索引的原因有两个：

- 确保一个或多个列中值的唯一性。
- 提高对表进行查询的效率。当执行查询，想以更快的速度找到所需的记录，或者想对表里的数据进行排序查询时，DB2 优化器会选择使用合适的索引。

如果表上不存在索引，那么必须对 SQL 查询中引用的每个表执行全表扫描。表中记录数越多，每条记录越长，全表扫描花费的时间越长，因为全表扫描需要顺序访问表中所有的行。虽然对于需要表中大多数记录的查询来说，使用表扫描效率可能更高，但对于只返回表中部分记录的查询，索引扫描可以更有效地访问表中的数据。

如果在 SELECT 语句中引用了索引列且优化器估计索引扫描比全表扫描快，那么优化器选择索引扫描。索引对应的数据页相对较少，因为一般情况下，索引都只包含表的部分列，因此读取它所需的时间比读取整个表所需的时间要少，尤其在表很大时更是如此。

如果对输出的排序需求可以与索引列相匹配，那么查询时将不需要再次对表中的数据进行排序。

每个索引条目的包含搜索键值和指向包含该值的行的指针。如果在 CREATE INDEX 语句中指定了 ALLOW REVERSE SCANS 参数，那么可以按升序和降序搜索这些值。

除搜索键值和行标识外(row id)，索引还可以包含非索引列。这样的列有可能使优化器仅从索引获取所需要的信息，而不必访问表本身。

注意：

要查询的表上存在索引并不保证结果集已排序，只有 ORDER BY 子句才确保结果集的排序。

尽管索引能显著缩短访问时间，但在少数情况下，它们也可能会给性能带来负面影响，所以要避免创建没意义的索引，在创建索引之前，考虑多个索引给磁盘空间和处理时间带来的影响。

每个索引都需要存储器或磁盘空间。准确的容量取决于表中的记录数以及索引包含的列的大小和数目。

对表执行的每个 INSERT 或 DELETE 操作都需要对表上的每个索引进行额外的更新。对于更改索引键值的每个 UPDATE 操作，也是如此。

LOAD 实用程序重建任何现有的关系索引或追加至现有的关系索引之后。可在 LOAD 命令中指定 MODIFIED BY indexfreespace 参数以覆盖创建索引时使用的索引 PCTFREE 值。

每个关系索引都有可能对 SQL 查询添加备用访问路径以供优化器考虑，这会增加编译时间，因此需要谨慎选择索引来满足应用程序的需要。

注意：

关系索引是相对于 XML 索引而言的，所以关系索引就是常规索引，也就是通常意义上的索引。

6.1.2　索引类型

1. 唯一索引与非唯一索引

索引可以定义为唯一的或非唯一的。非唯一的索引允许重复的键值，唯一的索引只允许某种键值的组合出现一次。虽然 NULL 值代表未知，但唯一索引只允许出现一次 NULL 值。索引是使用 CREATE INDEX 语句创建的。为支持主键或唯一约束，当创建主键或唯一键时隐式地创建索引。当创建唯一索引时，会检查表中键数据的唯一性，如果发现重复的键数据，索引创建将失效。索引可以创建为升序、降序或双向。选择哪个选项取决于应用程序如何访问数据。

2. 集群索引

我们可以创建集群索引来为表中的行排序，并且是按照所需结果集的物理顺序来排序。集群索引可以用 CREATE INDEX 语句的 CLUSTER 选项来创建。为获得最佳性能，应该在那些小型的数据类型(比如整型和 char(10))、具有唯一性的列，以及在范围搜索中经常要用到的列上创建集群索引。

集群索引允许对数据页采用更线性的访问模式，这将会大大提高范围性谓词查询的效率。但同时意味着插入操作要花更多的时间，因为不仅要维护索引的顺序，还要维护数据的顺序。当使用集群索引时，应考虑将数据页和索引页上的空闲空间增加为大约 15 到 35(而不是 PCTFREE 的默认值 10)，进而允许大容量的插入。

集群索引的好处包括：

- 在每个数据分区内，数据页以键的顺序排列。
- 集群索引改善了以键的顺序扫描整张表的性能。这是因为扫描访存第一页的第一行，然后访存同一页上的每一行，在访存了该页上的所有行之后，才移至下一页。这表示任何给定时间内都只需要表的一页位于缓冲池中。相反，如果表未集群，那么访存的每行有可能是不同页中的。除非缓冲池中有空间保存整个表，否则这会导致每页被访存多次，从而极大地减慢扫描速度。

6.2　索引结构

1. 标准表的表和索引管理

图 6-1 显示了标准表的逻辑表、记录和索引结构。

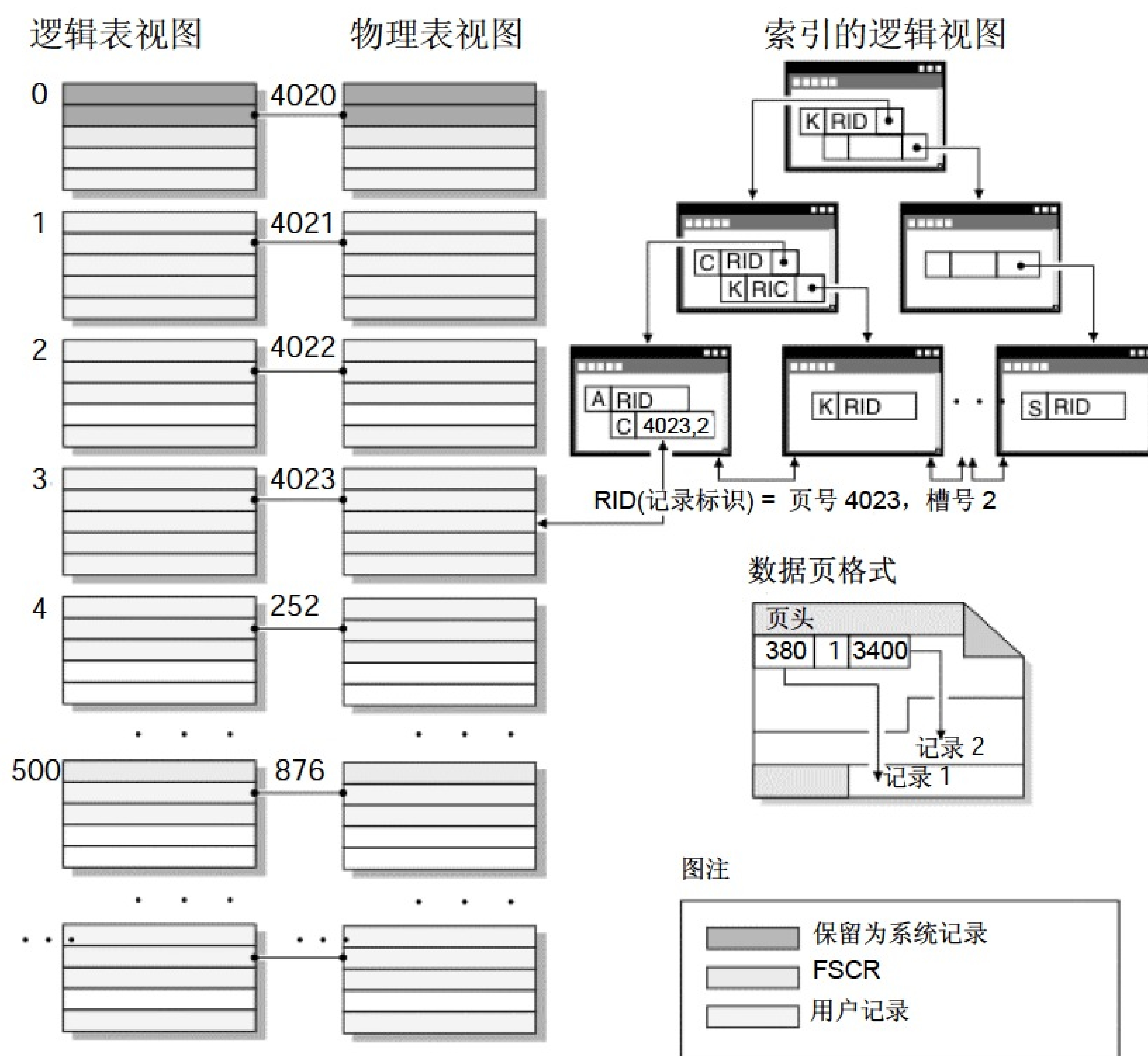


图 6-1　标准表的逻辑表、记录和索引结构

在标准表中，数据在逻辑上是按数据页的列表来组织的。这些数据页根据表空间的扩展数据块大小在逻辑上分组在一起。例如，如果扩展数据块(extent)大小是 4，第 0 至 3 页是第 1 个扩展数据块，那么第 4 至 7 页就是第 2 个扩展数据块，依此类推。

根据数据页大小以及记录大小的不同，每个数据页中所包含的记录数可能会有所变化。大多数页仅包含用户记录。但是，少数页包括特殊的内部记录，DB2 使用这些记录来管理表。例如，在标准表中，每个第 500 个数据页就有一个"空闲空间控制记录"(FSCR)。这些记录映射下面 500 个数据页(直到下一 FSCR 为止)上可供新记录使用的可用空间。当向表插入记录时，将使用这部分可用空间。

在逻辑上，索引页组织成 B+树，这可以有效地在表中定位带有给定键值的记录。索引页上的项数不是固定的，依赖于键的大小。对于 DMS 表空间中的表，索引页中的记录标识(RID)使用相对表空间页号，而不是对象相对页号。这使索引扫描能够直接访问数据页，而不需要"扩展数据块映像页"(EMP)来进行映射。

每个数据页都具有相同的格式。每个数据页的开头都有页头。在页头后面，有槽目录。槽目录中的每一条目都与该页中的一个记录相对应。条目本身是数据页中记录开始位置的字节位移。值为 - 1 的条目与已删除的记录相对应。

2. 记录标识和数据页

记录标识(RID)由页号及随后的槽号组成。在使用索引来标识 RID 之后，便使用 RID 来到达正确的数据页及该页上正确的槽号，如图 6-2 所示。一旦对记录指定了 RID，在进行表重组之前，该 RID 便不会更改。

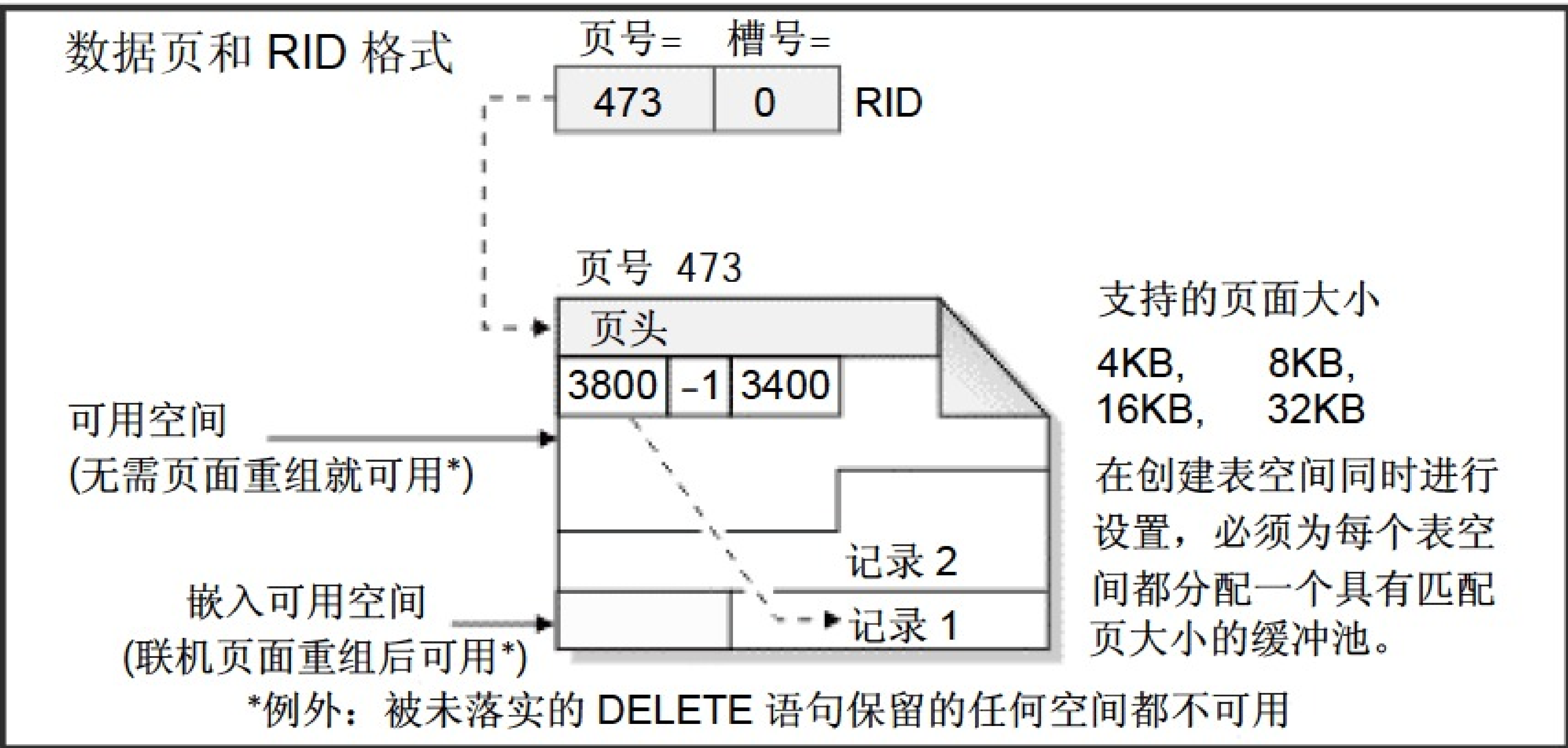


图 6-2　数据页和记录标识(RID)格式

重组表时，删除记录后，实际在页上留下的嵌入可用空间被转换成可使用的可用空间。根据记录在数据页上的移动重新定义 RID，以利用可使用的可用空间。

DB2 支持不同的页大小。对于有可能连续访问行的工作负载，请使用较大的页大小。例如，OLAP 应用程序或大量使用临时表的场合使用的便是顺序访问。对于更有可能进行随机访问的工作负载，使用较小的页大小。例如，OLTP 环境中使用的便是随机访问。

3. B+树结构

DB2 数据库管理器使用 B+树结构(Oracle 数据库有位图索引，而 DB2 没有)进行索引存储。B+树有一层或多层，如图 6-3 所示。其中，RID 表示行标识。

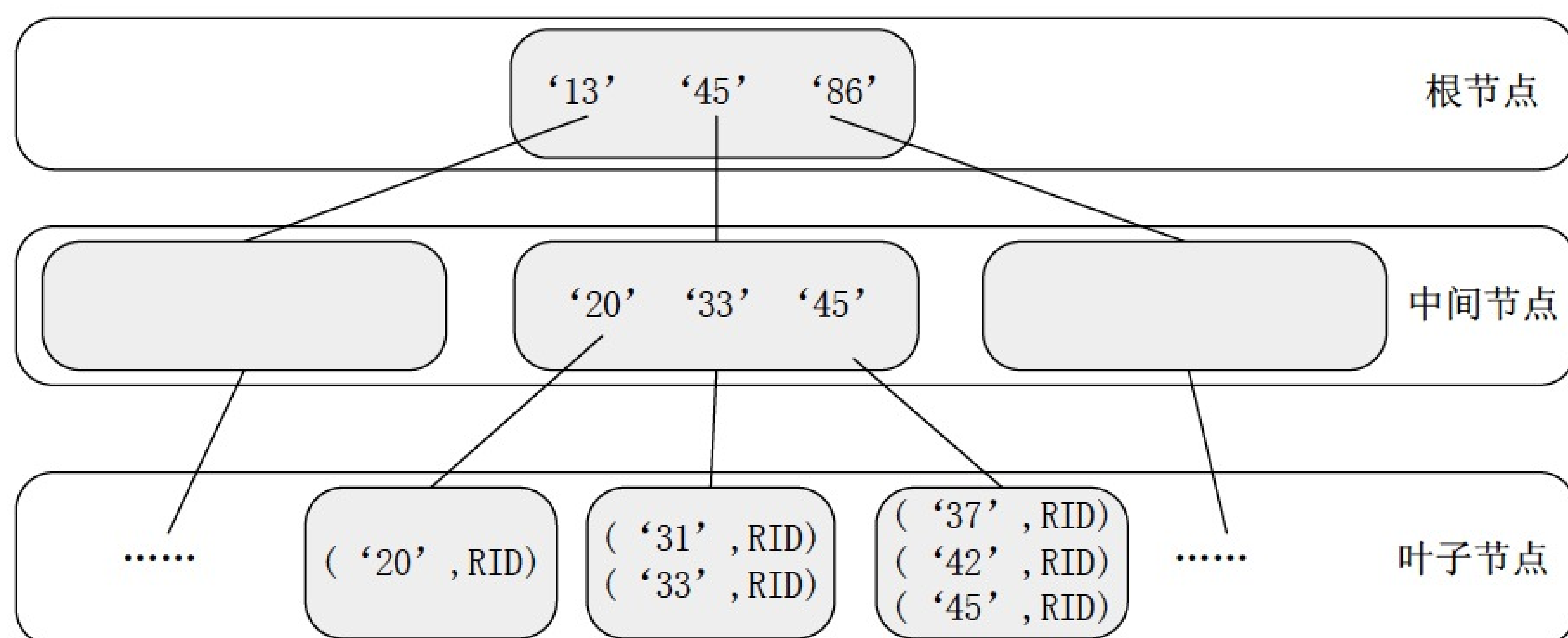


图 6-3　B+树结构

顶层称为根节点。底层由叶节点组成，底层存储的指针(RID)指向包含键值的表中的行。根节点层和叶节点层之间的那些层称为中间节点层。

当查找特定的索引键值时，索引管理器会从根节点开始搜索该索引树。对于下一层的每个节点，根都包含一个键。每个键的值是下一层中对应节点的最大现有键值。例如，如果索引有三层(图 6-3 中所示)，那么要查找某个索引键值，索引管理器将搜索根节点，以查找大于或等于要查找的键的第一个键值。根节点键指向特定的中间节点。索引管理器遵循此过程遍历中间节点，直到找到包含需要的索引键的叶节点为止。

图 6-3 显示要查找的键是“31”。在根节点中大于或等于“31”的第一个键是“45”，它指向下一层的中间节点。在该中间节点中大于或等于“31”的第一个键是“33”，它指向“33”的索引键及其对应的行标识的特定叶节点。然后扫描这个叶子节点中所有的记录，找到“31”键值对应的 RID，然后直接定位到具体的数据页。叶节点层也可以包含指向前一个叶节点的指针。这些指针允许索引管理器在找到范围内的某个值时，按任一方向扫描叶节点以检索某个范围内的值。仅当使用 ALLOW REVERSE SCANS 子句创建了索引时，才可能有按任意方向扫描的能力。

6.3　理解索引访问机制

DB2 优化器会首先考虑是否使用索引来实现查询。在优化器做出此决定之前，必须确定所查询的表上是否存在索引。可以查询任何表中的任何列，却不能期望每次都通过索引来做到这一点。所以，优化器必须能够访问未建立索引的数据，即全表扫描。

事实上，在大多数情况下，DB2 优化器会使用索引。因为索引可以大大提高数据检索的速度。然而，如果不存在索引，就无法使用它了。并且在某些情况下仅仅使用数据的全

扫描就可以极好地实现某些类型的 SQL 语句。例如，考虑下面这条 SQL 语句：

```
SELECT * FROM EMP;
```

在这条语句中，为什么 DB2 非要试图使用索引呢？这里没有 WHERE 子句，所以全扫描是最佳的。即使指定了 WHERE 子句，如果优化器判定全表扫描要比索引扫描效率更高，那么也不会使用索引。

存在索引的首要原因是可以改善性能，那为什么有时全表扫描会比索引扫描还要好呢？这是因为索引扫描可能比简单的表扫描要慢。例如，非常小的表可能只有几个页面。读取所有的页面可能比先读取索引页然后再读取数据页要快。甚至对于较大的表，在某些情况下，组织索引可能需要额外的 I/O 以实现查询。当不使用索引来实现查询时，产生的存取路径会采用表扫描方法。

表扫描通常会读取表中的每个页面。但在某些情况下，DB2 会非常聪明，它会限定要扫描的页面。此外，DB2 可以调用顺序预取以在请求某些页面之前就读取这些页面。当 SQL 请求需要按照数据存储在磁盘上的顺序来顺序地访问多行数据时，顺序预取特别有用。当优化器确定查询将按照顺序读取数据页面时，它会通知应该启用顺序预取。表扫描常常得益于顺序预取所做的提前读取工作，因为当某个查询请求数据时，这些数据已经放在内存中了。

快速的索引式访问

一般来讲，访问 DB2 数据的最快方式是使用索引。索引的目的就是能够快速准确地定位到具体的数据。图 6-3 显示了 B+树索引的结构。可以看出，通过简单地从树根遍历到叶子页，可以快速地找到相应的数据页，在那里有请求的数据。但是，DB2 采用的索引类型多种多样。所以，索引扫描有多种方式。

第一种(也是最简单的)索引扫描方式是直接索引查找。对于直接索引查找，DB2 从索引的根部开始，向下遍历，经过中间叶子页直到抵达相应的叶子页。在那里，它将读取实际数据页面的指针。根据索引条目，DB2 将读取正确的数据页面以返回期望的结果。对于 DB2，为了执行直接索引查找，在索引中必须为每个列提供值。例如，考虑 EMPLOYEE 表，该表有一个关于 DEPTNO、TYPE 和 EMPCODE 列的索引。现在考虑如下查询：

```
SELECT   FIRSTNAME, LASTNAME
FROM     EMPLOYEE
WHERE    DEPTNO = 5
         AND     TYPE = 'X'
         AND     EMPCODE = 10;
```

如果只指定这些列中的一列或两列，就不可能采用直接索引查找，因为 DB2 没有针对每列的值，不可能匹配整个索引关键字。虽然这样不能使用直接索引查找，但可以使用索引扫描。有两类索引扫描：匹配索引扫描和非匹配索引扫描。有时称匹配索引扫描为绝对定位；称非匹配索引为相对定位。还记得前面讨论的表扫描吗？索引扫描与此类似。在索引扫描中，按顺序读取索引的叶子页。

匹配索引扫描从索引的根页开始，遍历至叶子页，这种扫描方式与直接索引查找方式完全一样。然而，因为无法使用完整的索引关键字，所以 DB2 必须使用它所拥有的值来扫描叶子页，直到检索出所有匹配的值。现在考虑重写前面那个查询，这次没有使用 EMPCODE 谓词：

```
SELECT   FIRSTNAME, LASTNAME
FROM     EMPLOYEE
WHERE    DEPTNO = 5
         AND     TYPE = 'X';
```

通过从根部开始遍历索引，匹配索引扫描用相应的 DEPTNO 和 TYPE 值来查找第一个叶子页。但可能有多条索引条目具有这两个值的组合，而这些索引条目的 EMPCODE 值却不同。所以会按顺序扫描至右边的叶子页，直到不再遇到有效的 DEPTNO、TYPE 和各种 EMPCODE 的组合。

要请求执行匹配索引，必须指定索引关键字中的高次序列，就是前面示例中的 DEPTNO。这向 DB2 提供了遍历索引结构的起始点，从根页开始遍历，直到相应的叶子页。但如果没有指定这个高次序列，会发生什么呢？对上面的样本查询做点改动，不指定 DEPTNO 谓词：

```
SELECT   FIRSTNAME, LASTNAME
FROM     EMPLOYEE
WHERE    TYPE = 'X'
         AND     EMPCODE = 10;
```

在这个实例中，会用到非匹配索引扫描。在这种情况下，DB2 不能使用索引树结构，因为关键字中的第一列不可用。非匹配索引扫描从索引中的第一个叶子页开始遍历，应用可用的谓词，顺序扫描后续的叶子页。不使用根页和任何中间叶子页。

一种特殊类型的索引扫描是“仅索引访问”(index only access)。如果需要的全部数据都位于索引中，那么 DB2 完全可以避免读取数据页。例如：

```
SELECT   DEPTNO, TYPE
FROM     EMPLOYEE
WHERE    EMPCODE = 10;
```

我们这个数据库中包含 DEPTNO、TYPE 和 EMPCODE 列的索引。在前面的查询中，只请求查询这几列。所以，DB2 完全不需要访问表，因为在索引中可以找到所有数据。

DB2 可使用的另一类索引式访问是多索引访问。针对存取路径，多索引访问将使用多个索引。例如查询 EMPLOYEE 表，其中只有两个索引——关于 EMPNO 列的 IX1 和关于 DEPTNO 列的 IX2。然后，要求这条查询显示在某个特定部门工作的员工：

```
SELECT   LASTNAME, FIRSTNME, MIDINIT
FROM     EMPLOYEE
WHERE    EMPNO IN ('000100', '000110', '000120')
         AND      DEPTNO = 5;
```

DB2 会使用用于 EMPNO 谓词的 IX1 还是使用用于 DEPTNO 谓词的 IX2？为什么不一起使用这两者呢？这就是多索引访问的实质所在。根据谓词是用 AND 连接还是用 OR 连接，可将多索引访问分为两类：IndexANDING 和 IndexORING。IndexANDING 是先使用索引 IX1 和 IX2 取到索引扫描的结果，然后对两个扫描结果取交集；而 IndexORING 是先使用索引 IX1 和 IX2 取得索引扫描结果后，然后执行合并操作。

6.4　索引设计

创建索引时需要考虑以下一些注意事项：

- 不创建冗余的索引。例如：INX1=(a，b)和 INX2=(a，b，c)，INX1 就是冗余索引。INX1 是不必要的，因为 INX2 的节点中包含了 INX1 的所有键值和 RID。
- 经常观察索引的使用率。确保创建的索引保持较高的使用率，6.7.3 节会讲到如何查找数据库中使用率低下的索引。
- 考虑索引的维护成本。当向表中增加和删除数据时，这个表上所有的索引也会同时被维护，或者修改的字段被索引时，也会同时更新索引上的信息。所以如果表数据经常被更新，请考虑额外的索引会对性能产生什么样的影响。由于索引是键值的永久列表，它们在数据库中需要占用空间，因此创建许多索引就需要数据库中有更多的存储空间。所需的空间总量由键列的长度决定。
- 需要考虑索引列的顺序。按适当的列的顺序定义索引，该索引将大大提高查询的性能。

6.4.1　创建索引

使用命令行处理器来创建索引，可输入以下形式的语句：

```
CREATE INDEX <name> ON <table_name>(<column_name>)
```

可以创建索引，它将允许重复值，即非唯一索引。这允许在构成索引的一列或多列中存在重复值。下列 SQL 语句根据 EMPLOYEE 表中的 LASTNAME 字段创建名为 LNAME 并且按照升序排序的非唯一索引：

```
CREATE INDEX LNAME ON EMPLOYEE(LASTNAME ASC)
```

以下 SQL 语句基于电话号码列创建倒序的唯一索引：

```
CREATE UNIQUE INDEX PH ON EMPLOYEE(PHONENO DESC)
```

唯一索引确保在构成索引的一列或多列中不存在重复的值。如果构成索引的一个或多个列组成的集合已经有重复的值，唯一索引将创建失败。

关键字 ASC 按照列的升序放置这些索引项，而 DESC 则按照列的降序放置它们。默认为升序。

6.4.2 创建集群索引

以下 SQL 语句在 EMPLOYEE 表的 LASTNAME 列上创建集群索引，称为 INDEX1：

```
CREATE INDEX INDEX1 ON EMPLOYEE(LASTNAME) CLUSTER
```

为了让语句更有效，可以通过 PCTFREE 参数来创建集群索引，以便可以将新数据插入到正确的页上，从而维护该集群的次序。通常情况下，表上的 INSERT 操作越多，为维护集群所需要的 PCTFREE 值就越大。因为这个索引确定数据在物理页上放置的次序，所以对任何特定的表都只能定义一个集群索引。集群索引创建后需要对表进行 REORG 才能使表中的数据顺序与索引一致。

另一方面，如果这些新行的索引关键字值总是新的大关键字值，那么表的集群特性将尝试把它们放到表的末尾。在这种情况下，将表设置为追加方式可能优于使用集群索引。可以执行如下命令来将表设置为追加方式：ALTER TABLE tabname APPEND ON。

注意：

当表上有集群索引时不能开启 APPEND ON，因为这两个的作用相矛盾，请读者想想为什么。

PCTFREE 也适用于缓解表中记录 UPDATE 后引起的“溢出行”。

6.4.3　创建双向索引

CREATE INDEX 语句中的 ALLOW REVERSE SCANS 子句创建的索引可以正向或反向扫描。例如下面的 SQL 语句：

```
CREATE INDEX iname ON tname (cname ASC) ALLOW REVERSE SCANS
```

在这种情况下，基于给定列(cname)中的递增值(ASC)形成索引(iname)。尽管列上的索引定义用来按照递增次序扫描，但由于指定了 ALLOW REVERSE SCANS 子句，因而可以按照降序(反向)扫描。

索引页的合并与分裂

CREATE INDEX 语句的 MINPCTUSED 子句指定在索引叶页上最小已用空间的阈值。当创建索引时指定这个子句，那么每当从这个索引的叶子页中删除关键字后，并且叶子页上已用空间的百分比小于指定的阈值时，就检查相邻的索引叶子页来确定是否可以将两个叶子页上的关键字合并到单个索引页中。例如，下列 SQL 语句将启用联机索引合并：

```
CREATE INDEX LASTN ON EMPLOYEE(LASTNAME) MINPCTUSED 20
```

我们知道，每当从 EMPLOYEE 表中删除一条记录时，LASTN 索引将会删除相应的键，那么 MINPCTUSED 20 的意思就是说，删除索引里的键时，如果这个索引页上的已用空间占用整个索引页 20%或更小，就会尝试将这个索引页的键与相邻索引页合并，然后删除这个索引页。

图 6-4 是设置了 MINPCTUSED 后索引在线合并的示意图。

CREATE INDEX 语句的 PCTFREE 子句指定创建索引时，每个索引页中要留作空闲空间的百分比。在索引页上保留更多的空闲空间将导致更少的页分割，这将减少为重新获得顺序索引页面而重组表的需要，从而增加预存取，而预存取是可以提高性能的重要部件。此外，如果总是存在大关键字值，那么就要考虑降低 CREATE INDEX 语句的 PCTFREE 子句的值。

对于只读表上的索引，使 PCTFREE 为 0；对于其他索引，使 PCTFREE 为 10，提供可用的空间，从而加快插入操作的速度。此外，对于有集群索引的表，这个值应该更大一些，确保集群索引不会被分成太多的碎片。如果存在大量的插入操作，那么使用 15 到 35 之间的值或许会更合适一些。

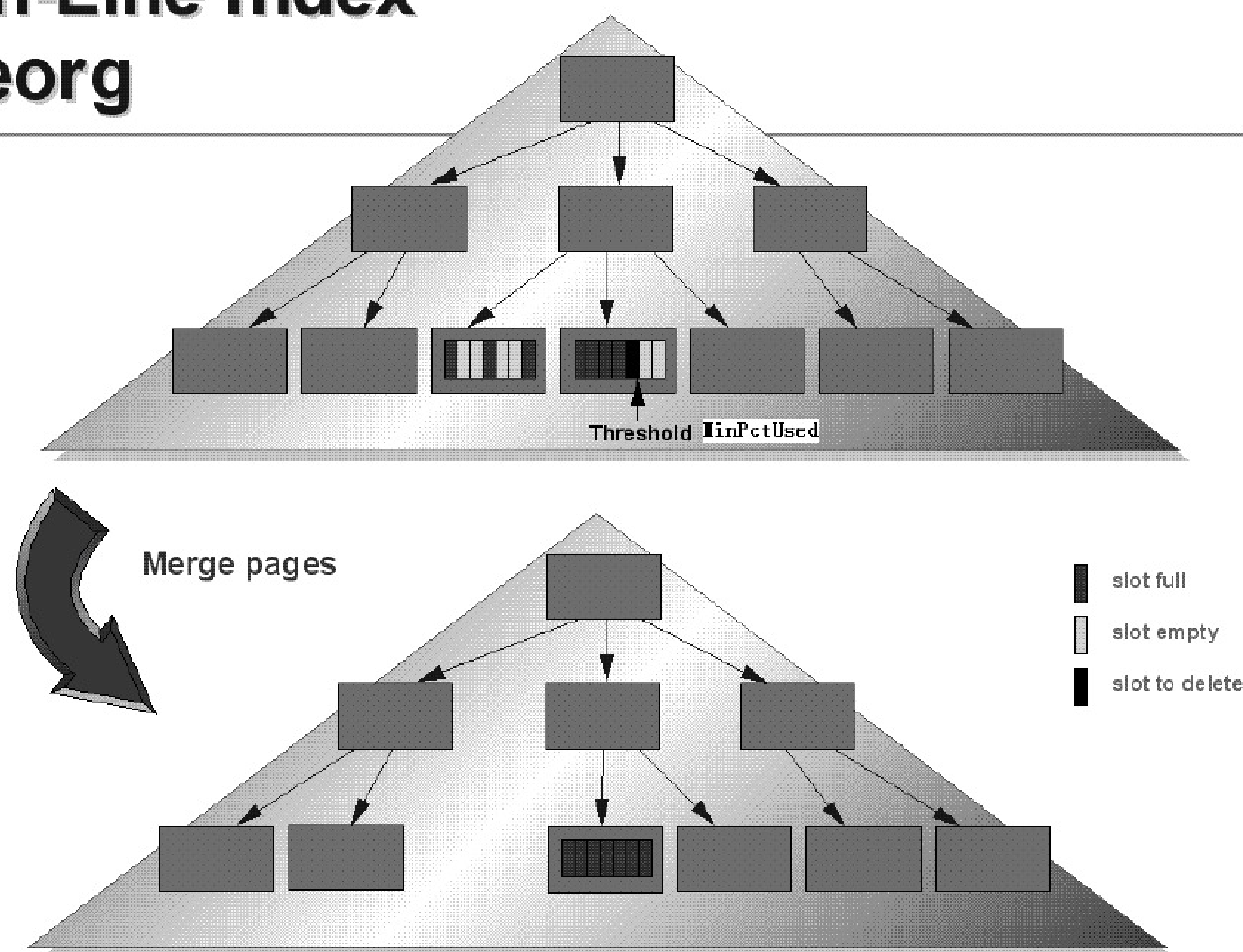


图 6-4　索引的在线重组

6.4.4　完全索引访问

CREATE INDEX 语句的 INCLUDE 子句指定在创建索引时，可以选择包含附加的列数据，这些附加的列数据将与键存储在一起，但实际上它们不是键自身的一部分，所以不被排序。在索引中包含附加列的主要原因是为了提高某些查询的性能。DB2 将不需要访问数据页，因为索引页早已经提供了数据值。只有定义唯一索引时才可以指定 INCLUDE 参数。

假设我们经常需要获得按 EMPNO 排序的员工列表。查询将如下所示：

```
SELECT EMPNO,EMPNAME FROM EMP ORDER BY EMPNO
```

下面的语句创建包含 INCLUDE 参数的唯一索引：

```
CREATE UNIQUE INDEX IEMPNO ON EMPNO(EMPNO) INCLUDE(EMPNAME)
```

这样，查询所需的所有数据都存在于索引中，不需要检索数据页。那么，为什么不干脆在索引中包括所有列的数据呢？首先，这需要数据库中的更多物理空间，因为本质上只是在索引中存放包含的数据。其次，只要更新了数据的值，数据的所有副本都需要更新，在经常需要更新和插入的数据库中，这是一项很大的开销。最后，索引查询之所以比表查

询更快，是因为一般来说，索引只包含表中的少数字段，因此索引页中包含更多的条数，遍历起来也占用更少的内存，随着索引字段的增加，这条特性也会慢慢消失。

6.4.5　与创建索引相关的问题

下面是创建索引时应该考虑的问题：

- 如果查询在可以接受的时间内完成，应避免添加索引，因为索引会降低更新操作的速度并消耗额外的空间。
- 重复率很高的列不适合建立索引。
- 考虑到管理上的开销，尽量使索引包含的列低于 5 个。
- 对于多列索引，将查询中引用最多的列放在定义的前面，范围查询的字段放在索引定义的最后。
- 提高索引的复用率。尽量用单个索引满足多条查询。避免添加与已有索引相似的索引，因为这样会给优化器带来更多的工作，并且会降低更新操作的速度。相反，我们应该修改已有的索引，使其包含附加的列。例如，假设在表的(c1，c2)上有索引 INDEX1。可以注意一下查询中使用了"where c2=?"，于是又创建了(c2)上的索引 INDEX2。但是这个索引没有添加任何东西，它只是 INDEX1 的冗余，现在反而成了额外的开销。
- 对于联机事务处理(OLTP)环境，创建一个或两个索引；对于只读查询环境，可以创建 5 个以上索引；对于混合查询和 OLTP 环境，可以创建 2 到 5 个索引。
- 要提高对父表执行删除和更新操作的性能，在外键上创建关系索引。

6.4.6　创建索引示例

在最频繁的查询和事务的 WHERE 子句中使用的那些列上创建关系索引。

以下 WHERE 子句：

```
WHERE WORKDEPT='A01' OR WORKDEPT='E21'
```

通常将会从 WORKDEPT 列上的索引获益，除非 WORKDEPT 列包含许多重复值。

根据 WHERE 子句过滤字段的顺序来创建关系索引。索引不仅可以提高普通查询的速度，也可以提高排序查询的效率(DISTINCT、ORDER BY、GROUP BY 都是用排序实现的)。

以下示例使用 DISTINCT 子句：

```
SELECT DISTINCT WORKDEPT
FROM EMPLOYEE
```

数据库管理器可以使用 WORKDEPT 上定义为升序或降序的索引来提高此查询的效

率。该索引也可用于 GROUP BY 子句中，将值分组，如下所示：

```
SELECT WORKDEPT, AVERAGE(SALARY)
 FROM EMPLOYEE
GROUP BY WORKDEPT
```

也可以把所要查询的所有列一起定义到索引中，这样的查询属于完全索引查询，不需要访问数据表，效率很高。

例如，考虑下列 SQL 语句：

```
SELECT LASTNAME  FROM EMPLOYEE  WHERE WORKDEPT IN ('A00','D11','D21')
```

如果为 EMPLOYEE 表的 WORKDEPT 和 LASTNAME 列定义了关系索引，那么通过扫描索引而不扫描整个表可能会更有效地处理该语句。注意，因为该谓词基于 WORKDEPT，因此此列应是该关系索引的第一列。

使用 INCLUDE 列创建关系索引可改善表上索引的使用。使用上述示例，可将唯一关系索引定义为：

```
CREATE UNIQUE INDEX x ON employee(workdept) INCLUDE (lastname)
```

指定 lastname 为 INCLUDE 列而不是索引键的一部分，意味着 lastname 只存储在索引的叶子页上。

注意：

INCLUDE 只能用在创建唯一索引中。想优化这条语句我们还有另一个取巧的办法，利用索引的冗余，也就是 CREATE INDEX x ON employee(workdept,lastname)

6.5 索引创建原则与示例

6.5.1 索引与谓词

DB2 中存在 4 种类型的查询谓词：Range-delimiting、Index-sargable、Data-sargable 和 Residual。其中，sargable 是 search argument 的缩写，表示在确定查询条件时使用某个参数进行搜索。这 4 种谓词放在 where 子句中作为查询条件时，性能依次下降。按最佳性能到最差性能的顺序排列如下所示：

- 范围定界(Range-delimiting)谓词
- 索引控制(Index-sargable)谓词
- 数据控制(Data-sargable)谓词

● 保留谓词(Residual)

表 6-1 概述了各谓词类型的特征，后面会更详细地描述每个类别。

表 6-1　谓词类型的特征摘要

特　　征	谓 词 类 型			
	范围定界	索引控制	数据控制	保留
减少索引 I/O	是	否	否	否
减少数据页 I/O	是	是	否	否
减少内部传送的行数	是	是	是	否
减少合格行的数目	是	是	是	是

Range-delimiting 谓词是指那些在进行索引扫描时可以明确限定边界的谓词，它们为索引搜索提供了 start 和/或 stop 键值。Index-sargable 谓词不能限定索引扫描的范围，但由于谓词中的列是索引的一部分，因此谓词也可以从索引中遍历出来。

比如，索引 IDX1 定义如下：

```
INDEX IDX1: NAME ASC, DEPT ASC, MGR DESC, SALARY DESC, YEARS ASC
```

并且有下面的查询子句：

```
WHERE NAME = :hv1
      AND DEPT = :hv2
      AND YEARS > :hv5
```

其中“NAME = :hv1”和“DEPT = :hv2”属于 Range-delimiting 谓词，而“YEARS > :hv5”属于 Index-sargable 谓词。数据库管理器利用索引数据评估谓词而不需要读取基表。Index-sargable 谓词通过减少需要从表中读取的行数而减少了数据页的数量。这类谓词不能减少索引页的读取数量。

那些不能被索引管理器搜索的谓词，但是可以被数据管理服务搜索的谓词被称为 Data-sargable 谓词。这些谓词需要访问基表的行，如果需要，数据管理服务将获取需要的列来搜索谓词，如<>谓词、Like '%H'谓词。

再如，给定下面定义在 PROJECT 表上的索引：

```
INDEX IDX0: PROJNO ASC
```

同时给定下面的查询：

```
SELECT PROJNO, PROJNAME, RESPEMP
FROM PROJECT
WHERE DEPTNO = 'D11'
ORDER BY PROJNO
```

谓词“DEPTNO = 'D11'”就是 Data-sargable 谓词，因为它没有被索引。

Residual 谓词一般是指需要的 I/O 已经超越了简单基表访问，Residual 谓词可以是使用关联的子查询，也可以是使用 ANY、ALL、SOME、IN 的子查询，或是读取 LONG、VARCHAR、LOB 数据的谓词。

6.5.2 根据查询使用的列建立索引

建立索引是用来提高查询性能的最常用方法。对于某个特定的查询，可以为某个表中出现在查询中的所有列建立联合索引，包括出现在 SELECT 语句和条件子句中的列。但简单地建立覆盖所有列的索引并不一定能有效提高查询，因为在多列索引中列的顺序是非常重要的。这个特性是由索引的 B+树结构决定的。一般情况下，要根据谓词的选择度来排列索引中各列的位置，选择度大的谓词使用的列放在索引的前面，把那些只存在于 SELECT 语句中的列放在索引的最后。例如下面的查询：

例 6-1 索引中的谓词位置。

```
select add date
from temp.customer
where city = 'WASHINGTON'
and cntry_code = 'USA';
```

这样的查询可以在 temp.customer 上建立(city，cntry_code，add_date)索引。由于该索引包含了查询所需的所有列，因而此查询将不会访问 temp.customer 的数据页，而是直接使用索引页。对于包含多列的联合索引，索引树中的根节点和中间节点存储了多列的值的联合。这就决定了存在两种索引扫描。回到例 6-1 中的查询，由于此查询在新建索引的第一列上存在谓词条件，DB2 能够根据这个谓词条件从索引树的根节点开始遍历，经过中间节点，最后定位到某个叶子节点，然后从此叶子节点开始往后进行在叶子节点上的索引扫描，直到找到所有满足条件的记录。这种索引扫描就是我们前面所讲的匹配索引扫描(Matching Index Scan)。但是如果将 add_date 放在索引的第一个位置，而查询并不存在 add_date 上的谓词条件，那么这个索引扫描将会从第一个索引叶子节点开始，它无法从根节点开始并经过中间节点直接定位到某个叶子节点，这种扫描的范围扩大到了整个索引，我们称之为非匹配索引扫描(Non-Matching Index Scan)。图 6-5 显示了 DB2 根据不同索引生成的存取计划。

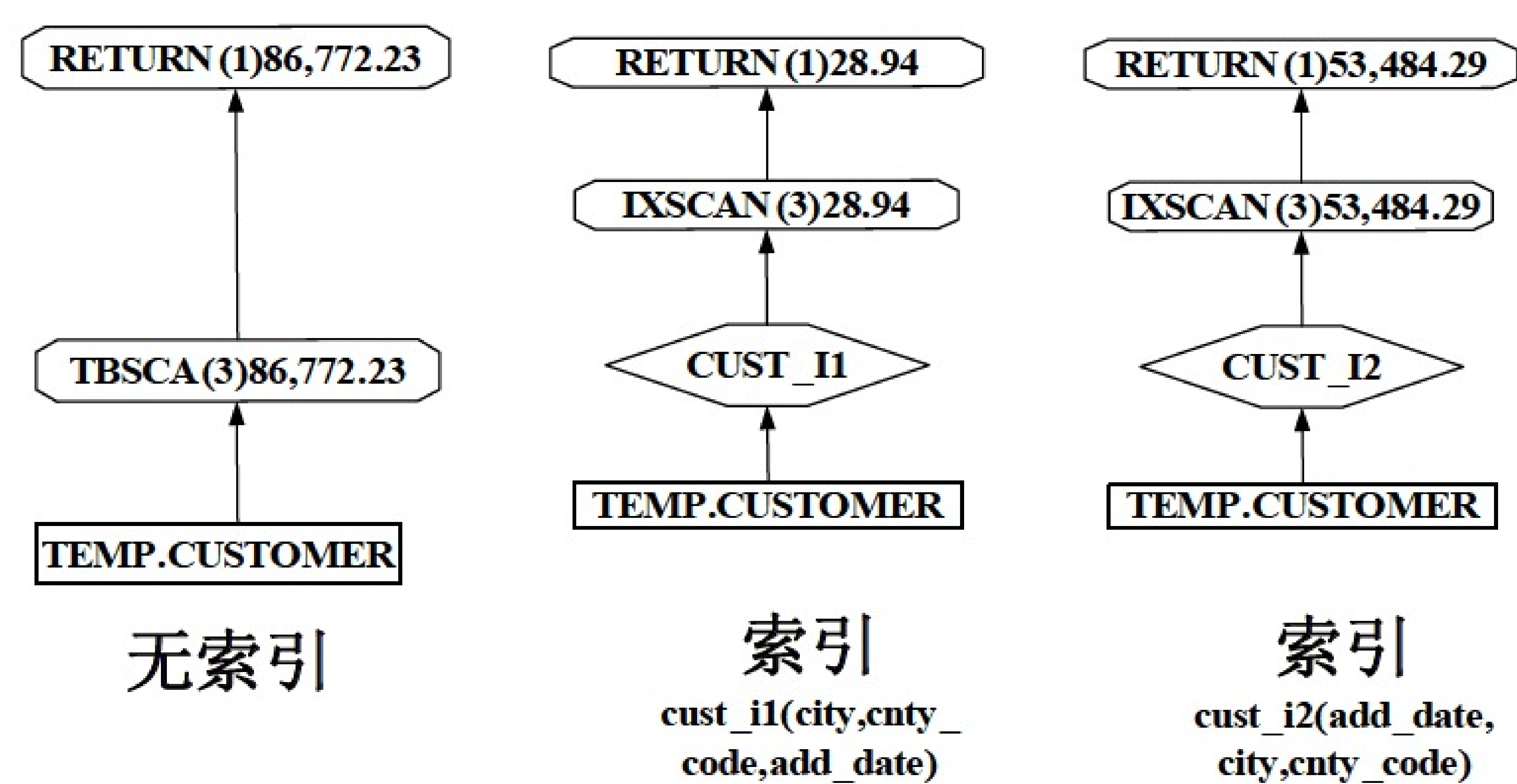


图 6-5　根据不同索引生成的存取计划

6.5.3　根据条件语句中谓词的选择度创建索引

因为建立索引需要占用数据库的存储空间，所以需要在空间和时间性能之间进行权衡。很多时候，只考虑在那些条件子句中有条件判断的列上建立索引会同样有效，同时节约了空间。例如例 6-1 中的查询，可以只建立(city，cntry_code)索引。我们还可以进一步地检查条件语句中这两个谓词的选择度，执行下面的语句检查谓词选择度：

例 6-2　检查谓词选择度。

```
查询：
1. select count(*) from temp.customer where city = 'WASHINGTON'
   and cntry_code = 'USA';
2. select count(*) from temp.customer where city = 'WASHINGTON';
3. select count(*) from temp.customer where cntry_code = 'USA';
结果：
1. 1404
2. 1407
3. 128700
```

选择度越大，过滤掉的记录越多，返回的结果集也就越小。从例 6-2 的结果可以看到，第二个查询的选择度几乎和整个条件语句相同。因此可以直接建立单列索引(city)，其性能与索引(city，cntry_code，add_date)相差不多。表 6-2 中对这两个索引的性能和大小进行了对比。

表 6-2　两个索引的性能和大小对比

索　　引	查询计划总代价	索 引 大 小
cust_i1(city，cntry_code，add_date)	28.94 timerons	19.52MB
cust_i3(city)	63.29 timerons	5.48MB

从表 6-2 中可以看出：单列索引(city)具有更佳的性能空间比。也就是说，占有尽可能小的空间而得到尽可能高的查询速度。

6.5.4　避免在建有索引的列上使用函数

这是一个很简单的原则。如果在建有索引的列上使用函数，由于函数的单调性不确定，函数的返回值和输入值可能不会一一对应，就可能存在索引中位置差异很大的多个列值可以满足带有函数的谓词条件，因此 DB2 优化器将无法进行 Matching Index Scan，更坏的情况下可能会导致直接进行表扫描。图 6-6 中对比了使用函数前后的存取计划的变化。

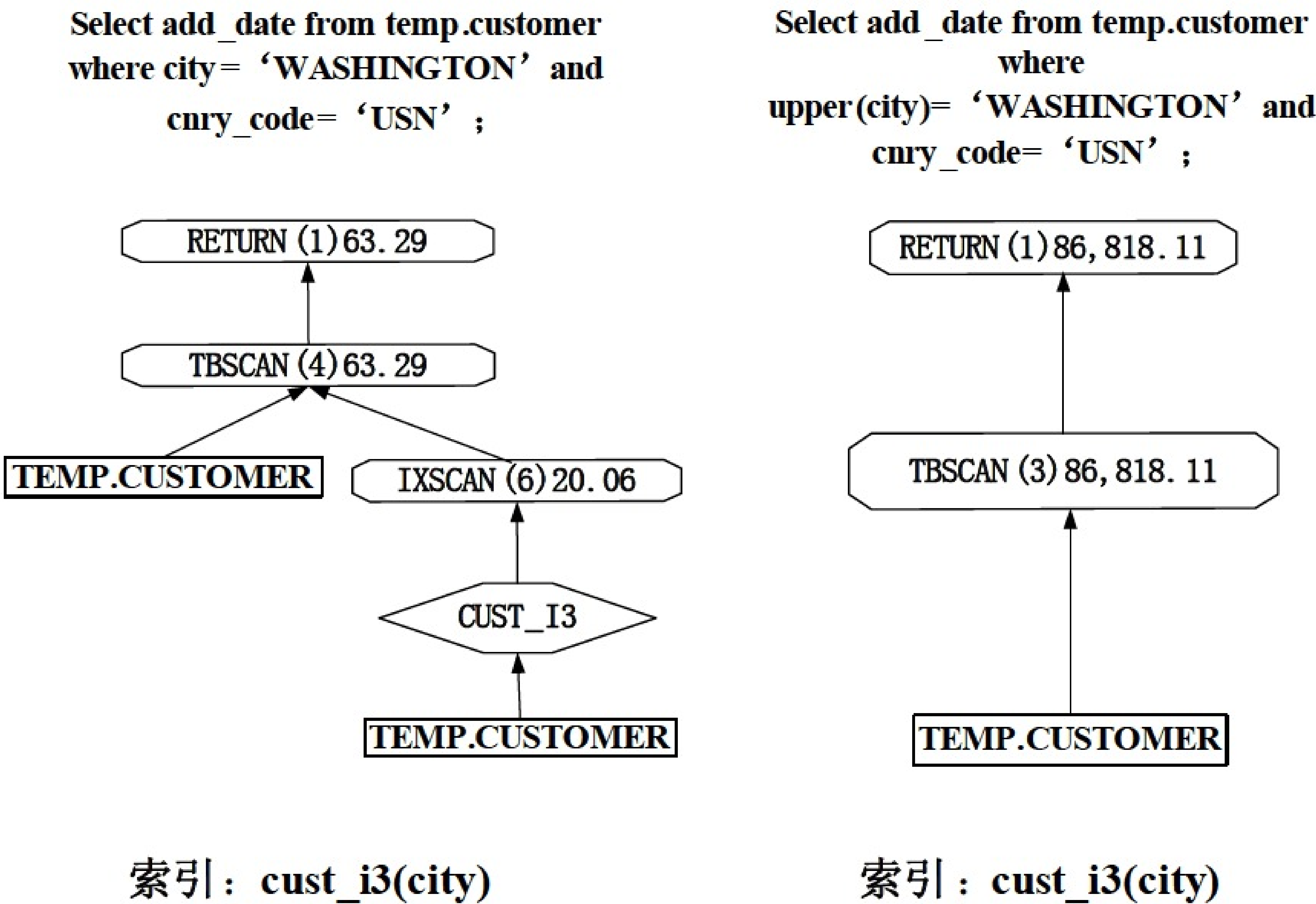


图 6-6　使用函数前后的存取计划的变化

6.5.5　在那些需要被排序的列上创建索引

这里的排序不仅仅指 ORDER BY 子句，还包括 DISTINCT、UNION 或 GROUP BY 子

句，它们都会产生排序操作。由于索引本身是有序的，在创建过程中已经进行了排序处理，因为根据需要排序的列来创建索引可以避免查询时再次对数据进行排序。这种情况一般针对没有条件语句的查询。如果存在条件语句，DB2 优化器会首先选择出满足条件的记录，然后才对中间结果集进行排序。对于没有条件语句的查询，排序操作在总的查询代价中会占有较大比重，因此能够较大限度地利用索引的排序结构进行查询优化。此时可以创建单列索引，如果需要创建联合索引，那么需要把被排序的列放在联合索引的第一列。图 6-7 对比了如下查询在创建索引前后的存取计划。

```
select distinct add_date from temp.customer;
```

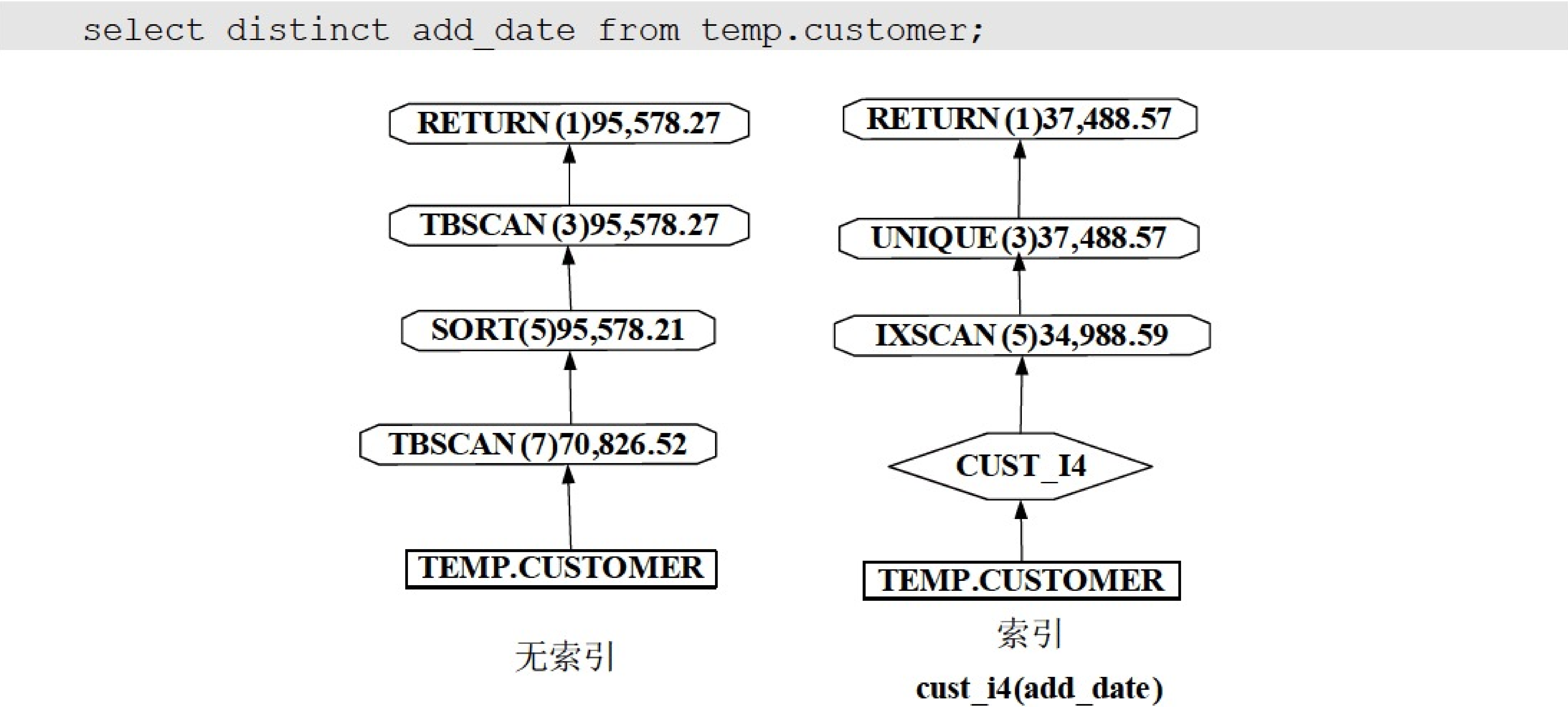


图 6-7　在创建索引前后的存取计划

从图 6-7 中我们可以看出，在没有索引的情况下 SORT 操作是 24751.69 timerons；但在有索引的情况下，不再需要对结果集进行排序，而是可以直接进行 UNIQUE 操作，图中显示这一操作只花费了 2499.98 timerons。

图 6-8 对比了以下查询在创建联合索引前后的存取计划，从中可以更好地理解索引对排序操作的优化：

```
select cust_name from temp.customer order by add_date;
```

索引的 B+树结构决定了索引 temp.cust_i5 的所有叶子节点本身就是按照 add_date 排序的，所以对于该查询而言，只需要顺序扫描索引 temp.cust_i5 的所有叶子节点。但是对于 temp.cust_i6 索引，所有叶子节点是按照 cust_name 排序的，因此在经过对索引的叶子节点扫描并获得所有数据之后，还需要对 add_date 进行排序操作。

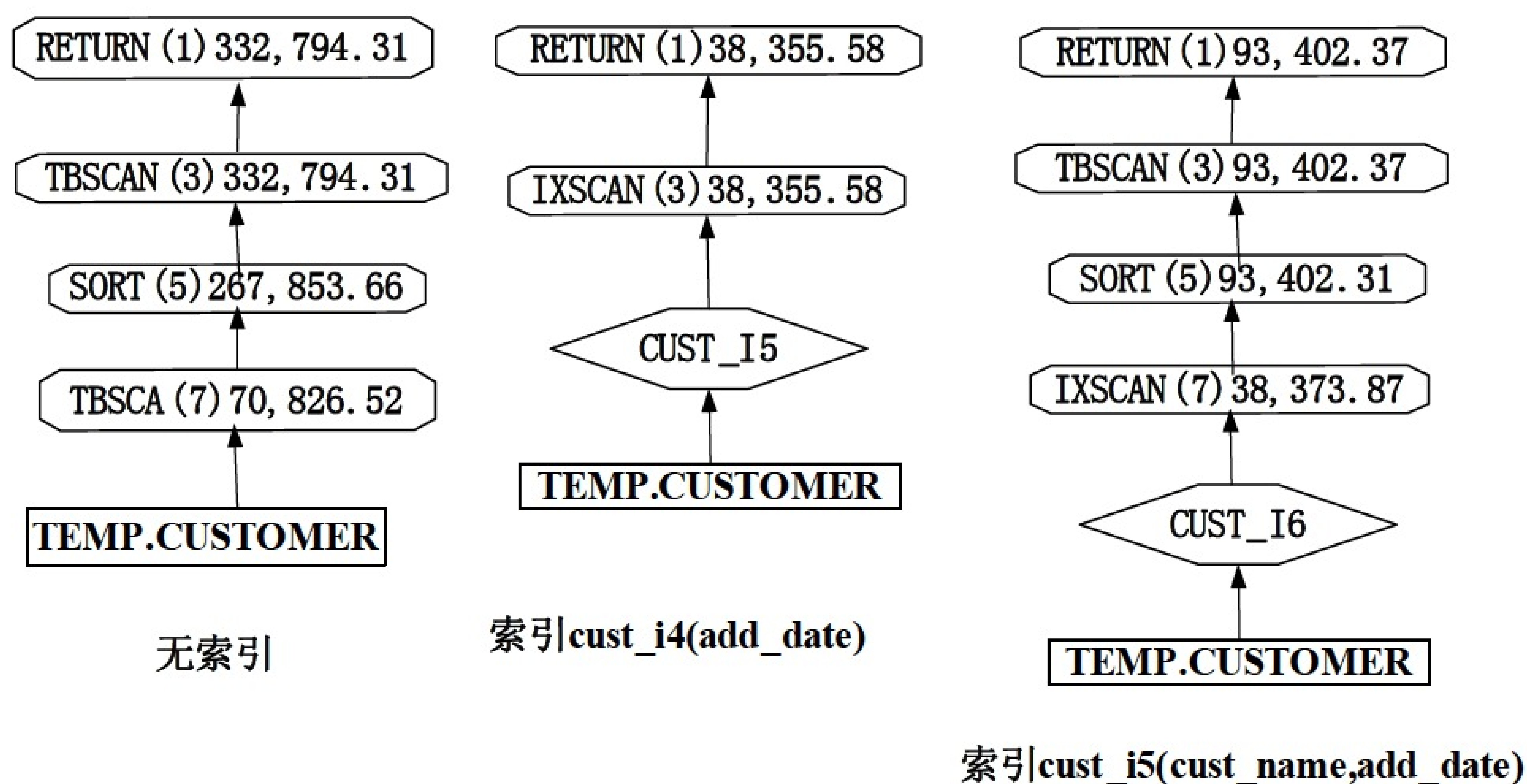


图 6-8　创建联合索引前后的存取计划

6.5.6　合理使用 INCLUDE 关键词创建索引

对于类似下面的查询：

```
select cust name from temp.customer
where cust_num between '0007000000' and '0007200000';
```

我们可以在 cust_num 和 cust_name 上建立联合索引来提高查询性能。但是由于 cust_num 是主键，是唯一的，因此可以使用 INCLUDE 关键字创建唯一索引：

```
create unique index temp.cust i7 on temp.customer(cust num)include
(cust_name);
```

使用 INCLUDE 后，实际上只是将 cust_name 列的数据从表中复制一份放到索引树的叶子节点，并不属于索引的键。在这种情况下，对于查询来说，INCLUDE 的唯一索引和联合索引是同等的；但是对于索引的生成时间以及今后维护的代价来说，创建带有 INCLUDE 列的唯一索引会带来优于联合索引的性能，因为 INCLUDE 列里的数据不参与排序。对于上面的查询示例，创建索引 temp.cust_i7 后存取计划的代价为 12338.7 timerons，创建联合索引 temp.cust_i8(cust_num，cust_name)后的代价为 12363.17 timerons。一般情况下，当查询的 WHERE 子句中存在主键的谓词时，我们就可以创建带有 INCLUDE 列的唯一索引，用完全索引访问来提高查询性能。

注意：

INCLUDE 只能用在创建唯一索引中。

6.5.7　指定索引的排序属性

下面是用来显示最近一个员工入职时间的查询：

```
select max(add_date) from temp.employee;
```

很显然，这个查询会进行全表扫描。查询计划如图 6-9(a)所示。

显然，我们可以在 add_date 上创建索引。根据下面的命令创建索引后的查询计划如图 6-9(b)所示。

```
create index temp.employee_i1 on temp.employee(add_date);
```

这里存在误区，大家可能认为既然查询里要取得的是 add_date 的最大值，而我们又在 add_date 上建立了索引，优化器应该知道从索引树中直接寻找最大值。但是实际情况并非如此，因为创建索引的时候并没有指定排序属性，默认为 ASC 升序排列，DB2 将会扫描整个索引树的叶子节点，在取得所有值后，取其中最大值。我们可以通过设置索引的排序属性来提高查询性能，根据下面的命令创建索引后的查询计划如图 6-9(c)所示。

```
create index temp.employee_i2 on temp.employee(add_date desc);
```

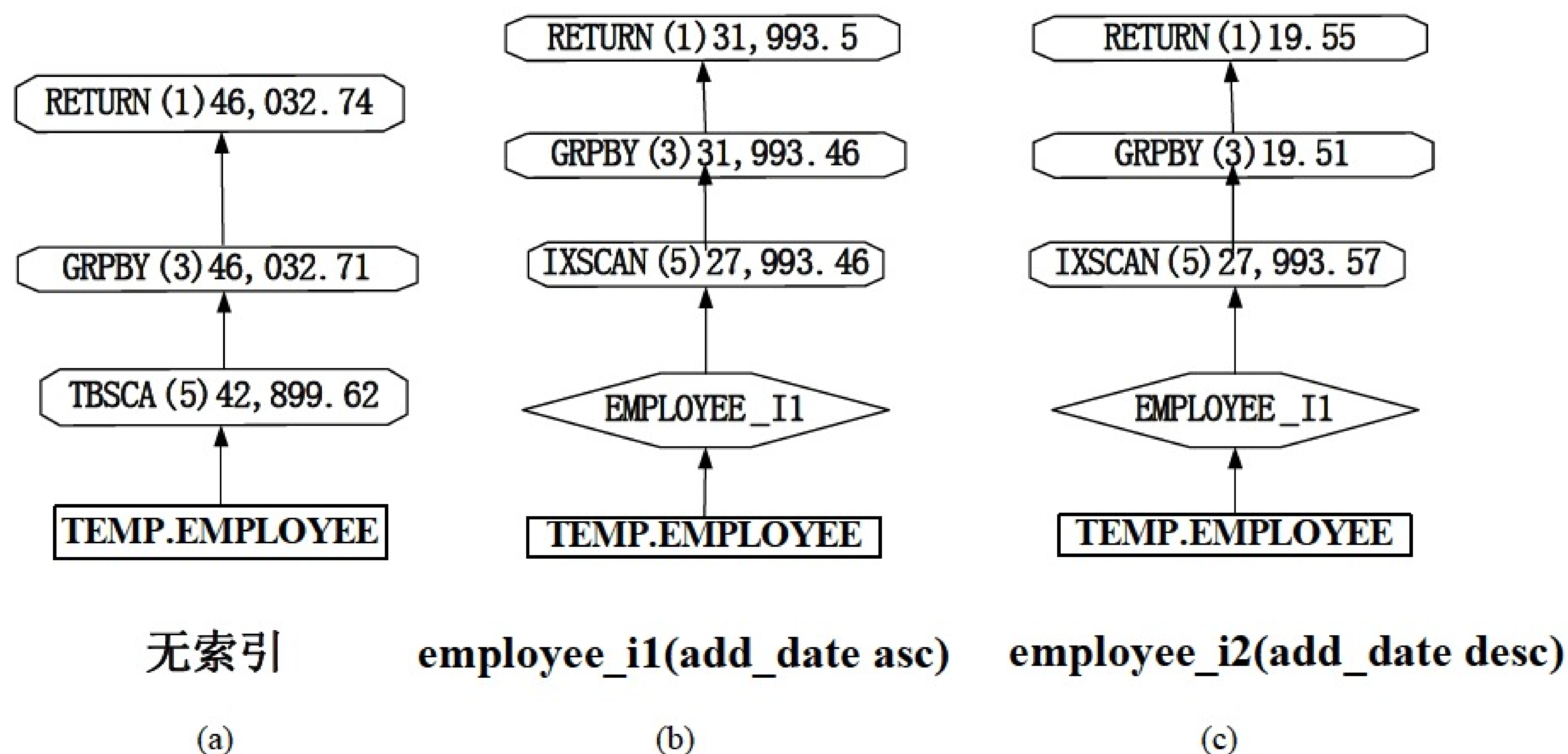


图 6-9　查询计划

对于降序排列的索引，DB2 不需要扫描整个索引树的叶子节点，因为第一个节点便是最大的。我们同样可以使用 ALLOW REVERSE SCANS 来指定索引为双向扫描，具有和

DESC 近似的查询性能。ALLOW REVERSE SCANS 可以被认为是 ASC 和 DESC 的组合，只是在以后数据更新的时候维护成本会相对高一些。

虽然无法改变索引的排序属性，但是我们具有额外的信息，该公司每个月都会有新员工入职，那么这个查询就可以改写成：

```
select max(add_date) from temp.employee
where add_date > current timestamp - 1 month
```

像这样通过限定查询范围也可以有效地提高查询性能。

6.6 影响索引性能的相关配置

6.6.1 设置影响索引性能的配置参数

1. util_heap_sz 配置参数

CREATE INDEX 和 REORG INDEXES 都支持其他用户或应用程序对基表进行写访问。如果在创建或重组索引时需要对基表执行大量更新活动，请考虑配置实用程序堆(util_heap_sz)。在同步更新阶段，实用程序堆将提高创建或重组索引的速度。对正在创建或重组的索引执行的所有写活动都记录在 DB2 日志和内部内存缓冲区空间中。内部内存缓冲区空间是根据需要从实用程序堆中分配的指定内存区域，用来存储对正在创建或重组的索引所做的更改。使用该内存区域可以使同步更新阶段能够更快地工作。在创建或重组操作完成后，将释放所分配的内存。如果能确保有足够的实用程序堆来容纳对正在创建或重组的索引进行的全部或大部分更改，那么会提高同步更新阶段的性能。

2. sheapthres 配置参数

每个子代理程序将获取 sortheap 配置参数指定的内存量，以便在扫描表时避免排序溢出。应监视排序溢出数并相应地增大 sheapthres。

6.6.2 为索引指定不同的表空间

可将索引存储在与表数据不同的表空间中。这样，通过减少索引访问期间读/写磁头的移动，可以更有效地使用磁盘存储器。也可以在更快的物理设备上创建索引表空间。而且，可以将索引表空间指定给不同的缓冲池，由于它们不与表数据页竞争，因此可以将索引页较长时间保存在缓冲区内。

当不将索引放置在单独的表空间中时，数据页和索引页使用相同的扩展数据块大小和

预存取。如果对索引使用不同的表空间，那么可为表空间的所有特征选择不同的值。因为索引通常比表小，并且分布在更少的容器中，因此索引通常只具有较小的扩展数据块大小，如 8 页和 16 页。当查询优化器选择访问方案时，它将考虑用于表空间的设备的速度。

6.6.3　确保索引的集群度

如果 SQL 语句需要排序(例如包含 ORDER BY、GROUP BY 和 DISTINCT 子句)，那么即使索引可以满足排序，但是在下列情况下，优化器可能不会选择索引：

- 索引集群程度较低。有关信息请查看 SYSCAT.INDEXES 的 CLUSTERRATIO 和 CLUSTERFACTOR 列。
- 由于表很小，因此扫描该表并在内存中对结果集进行排序时消耗的成本更低。

索引的集群比率和集群因子统计信息

索引的集群比率统计信息存储在 SYSTCAT.INDEXES 目录表的 CLUSTERRATIO 列中。此值(在 0 到 100 之间)表示使用索引的数据集群的程度。要将 CLUSTERFACTOR 与 CLUSTERRATIO 的值进行比较，可以将 CLUSTERFACTOR 乘以 100 以获得百分比。

注意：

通常，表中只有一个索引可以具有较高的集群度。索引扫描在具有较高集群比率的情况下可能执行得更好。低的集群率导致此类扫描要执行更多的 I/O，因为松散的集群率需要访问更多的数据页。增大缓冲区大小也可以提高非集群索引的性能。如果表数据最初是针对某个索引集群的，而集群统计信息指示现在的集群率很低，那么可能想重组该表以再次集群数据。

6.6.4　使表和索引统计信息保持最新

当创建新索引后，运行 RUNSTATS 实用程序来收集索引统计信息。这些统计信息使优化器能够确定使用索引是否能提高访问性能。

6.7　索引维护

在创建索引之后，经过大量的插入和删除操作以后，索引的性能会下降。考虑以下建议使索引尽可能小并效率高：

- 启用联机索引整理碎片
- 使用 MINPCTUSED 子句创建索引
- 如有必要，删除并重新创建现有的索引

- 保持索引的集群度

6.7.1 异步索引清除(AIC)

异步索引清除(AIC)是在使索引条目失效的操作之后的延迟索引清除。根据索引的类型，条目可以是行标识(RID)或块标识(BID)。无论是哪种标识，这些条目都将由索引清除程序(db2aic)除去，索引清除程序在数据库后台异步运行。

注意：
如果分区表定义了物化查询表(MQT)，那么 AIC 要在执行了 SET INTEGRITY 操作之后才启动。

当 AIC 正在进行时，将维护正常的表访问。访问索引的查询将忽略尚未清除的任何无效条目。

在大多数情况下，对与分区表关联的每个非分区索引启动清除程序。内部任务分发守护程序(db2taskd)负责将 AIC 任务分发给适当的数据库分区并指定数据库代理程序。

分发守护程序和清理代理程序都是内部系统应用程序。可以用 LIST APPLICATION SHOW DETAIL 来观察它们的执行情况，应用程序名称分别为 db2taskd 和 db2aic。为了防止意外中断，不能强制执行系统应用程序。只要数据库是活动的，分发守护程序就一直处于联机状态。清除程序在完成清理之前保持活动状态。如果在进行清理时取消激活数据库，那么当重新激活数据库时将继续进行 AIC。

1. AIC 性能

AIC 仅对性能产生很小的影响。

每个清除程序都获取最小表空间锁定(IX)和表锁定(IS)。如果清除程序确定其他应用程序正在等待这些锁定，就会释放这些锁定。如果发生这种情况，那么清除程序就会暂挂处理 5 分钟。

清除程序与实用程序具有集成的优先级设置。默认情况下，每个清除程序的实用程序的影响优先级为 50。可以使用 SET UTIL_IMPACT_PRIORITY 命令来更改此优先级。

2. AIC 监视

可以使用 LIST UTILITIES SHOW DETAIL 命令来监视 AIC。每个索引清除程序都作为单独的实用程序出现在监视器中。

以下示例显示了 AIC 活动：

```
    $ db2 list utilities show detail

    ID                            = 1489
    Type                          = ASYNCHRONOUS INDEX CLEANUP
    Database Name                  = SLOGDB
    Member Number                  = 0
    Description                    = Table: SLOGDB  .EN SERVICELOG, Index:
SLOGDB  .servicelog_stmp_index
    Start Time                     = 03/28/2017 11:05:34.450685
    State                         = Executing
    Invocation Type                 = Automatic
    Throttling:
       Priority                    = 50
    Progress Monitoring:
          Total Work               = 3537988 pages
          Completed Work            = 1394690 pages
          Start Time               = 03/28/2017 11:05:34.450924


    ID                            = 1488
    Type                          = ASYNCHRONOUS INDEX CLEANUP
    Database Name                  = SLOGDB
    Member Number                  = 0
    Description                    = Table: SLOGDB  .EN SERVICELOG, Index:
SLOGDB  .PK_SERVICELOG_ID
    Start Time                     = 03/28/2017 11:05:34.450697
    State                         = Executing
    Invocation Type                 = Automatic
    Throttling:
       Priority                    = 50
    Progress Monitoring:
          Total Work               = 13542316 pages
          Completed Work            = 486981 pages
          Start Time               = 03/28/2017 11:05:34.450953
```

在上述示例中，有两个清除程序正在对 SLOGDB.EN_SERVICELOG 表进行操作。一个清除程序正在处理索引 SLOGDB.servicelog_stmp_index，另一个清除程序正在处理索引 SLOGDB.PK_SERVICELOG_ID。进度监视部分显示需要清除的总索引页数和当前已完成的索引页数。

状态字段指示清除程序的当前状态。通常，状态为“正在执行”。如果清除程序正在

等待被指定给可用的数据库代理程序，或者如果清除程序由于锁定争用而临时暂挂，那么清除程序可能处于“正在等待”状态。

6.7.2 联机索引整理碎片

用户定义的索引叶子页上的最小已使用空间量的阈值(MINPCTUSED)将启用联机索引整理碎片。当从叶子页删除了索引键且超过阈值时，可检查相邻的索引叶子页以确定是否可合并两个叶子页。如果某一页上有足够的空间用于合并两个相邻的页，那么立即在后台合并。

联机索引整理碎片期间，不合并索引非叶子页。但是，会删除空的非叶子页以供相同表上的其他索引重用。要对 DMS 存储模型中的其他对象释放这些非叶子页，或者释放 SMS 存储模型中的磁盘空间，那么请对表或索引进行全面重组。表和索引的全面重组可以使索引尽可能地小。

要重组键标记为已删除但实际上仍在索引页中的 type-2 索引的碎片，可用 CLEANUPONLYALL 选项执行 REORG INDEXES 命令。CLEANUP ONLY ALL 选项整理索引碎片，与 MINPCTUSED 的值无关。用 CLEANUP ONLY ALL 执行 REORG INDEXES，如果两个相邻叶子页的合并至少可以在合并页上留下 PCTFREE 可用空间，那么合并这两个相邻叶子页。在索引创建时指定 PCTFREE，默认值为 10%。

6.7.3 查找使用率低下的索引

我们都知道多余的无用索引会影响表更新的性能，如何找出系统中使用率低下的索引呢？先看如下例子：

```
db2pd -d sample -tcbstats all tbspaceid=3 tableid=5
```

通过这条命令我们可以得到如下输出

```
    $db2pd -d sample -tcbstats all  tbspaceid=3 tableid=5

    Database Member 0 -- Database SAMPLE -- Active -- Up 0 days 00:29:58 -- Date
10/24/2012 10:35:58

    TCB Table Information:

    Address            TbspaceID TableID PartID MasterTbs MasterTab TableName
SchemaNm ObjClass DataSize  LfSize    LobSize   XMLSize

    0x07000000461D4AF0 3        5      n/a    3         5        TEST
DB2INST1 Perm     1          0          0          0
```

```
    TCB Table Stats:

    Address             TableName           SchemaNm Scans      UDI        RTSUDI
PgReorgs   NoChgUpdts Reads      FscrUpdates Inserts    Updates    Deletes
OvFlReads  OvFlCrtes  PgDictsCrt CCLogReads StoreBytes BytesSaved

    0x07000000461D4AF0 TEST               DB2INST1 0          0          0
0          0          0          0          0          0          0          0
0          0          0          -          -



    TCB Index Information:

    Address             InxTbspace ObjectID PartID TbspaceID TableID MasterTbs
MasterTab TableName          SchemaNm IID   IndexObjSize

    0x07000000461D5960 3          5        n/a    3         5       3         5
TEST              DB2INST1 1     4

    TCB Index Stats:

    Address             TableName           IID    PartID EmpPgDel    RootSplits
BndrySplts PseuEmptPg EmPgMkdUsd Scans      IxOnlyScns KeyUpdates InclUpdats
NonBndSpts PgAllocs   Merges     PseuDels   DelClean   IntNodSpl

    0x07000000461D5960 TEST               1     n/a    0          0          0
0          0          4          4          0          0          0          0          0
0          0          0

    RBPPTRN:/home/db2inst1$
```

可以看到这条命令的输出包括了这个表和这个表上所有索引的信息，以及表和索引上的扫描、更新、删除、插入的次数。由此可以判断，哪些索引上扫描的次数较少，就说明它们的利用率低下，然后可以考虑是否该将它们删除，或者通过下面的语句查看，输出可以根据需要来调整。

```
    db2 "select T.TABSCHEMA,T.TABNAME,S.INDNAME,S.COLNAMES,T.INDEX SCANS,
T.INDEX_ONLY_SCANS from TABLE(MON_GET_INDEX('','',-2)) as T ,SYSCAT.INDEXES AS
S WHERE T.TABSCHEMA = S.TABSCHEMA AND  T.TABNAME = S.TABNAME AND  T.IID = S.IID
AND T.TABSCHEMA NOT IN ('SYSIBM','SYSCAT','SYSIBMADM','SYSTOOLS','SYSSTAT',
'SYSPUBLIC') AND UNIQUERULE<>'P' ORDER BY T.TABNAME DESC with ur"
```

这条命令可以查看具体的表以及这个表上的索引信息，如果想查看数据库里所有表的索引信息，只用去掉 tbspaceid 和 tableid 即可：

```
$db2pd -d sample -tcbstats all
```

6.7.4 索引压缩

创建索引时，在默认情况下，索引压缩功能对已压缩的表处于开启状态，对未压缩的表处于关闭状态。可以通过显式的 create index 语句的 compress yes 子句，也可以通过 alter index 语句的 compress yes 子句来启用索引压缩，开启索引压缩后需要执行索引重组以便使压缩生效。

索引压缩实际上是根据数据库管理器所选择的压缩算法对索引页在磁盘和内存中的格式进行修改，以便减少存储空间的消耗。不同类型的索引以及索引包含数据的不同，具体的压缩程度也会有所变化。索引压缩和表压缩很相似，更详细的内容请参见《高级进阶 DB2(第 2 版)》中第 3 章的内容。

6.8 DB2 Design Advisor(db2advis)

DB2 Design Advisor 提供了强大的功能，如图 6-10 所示。

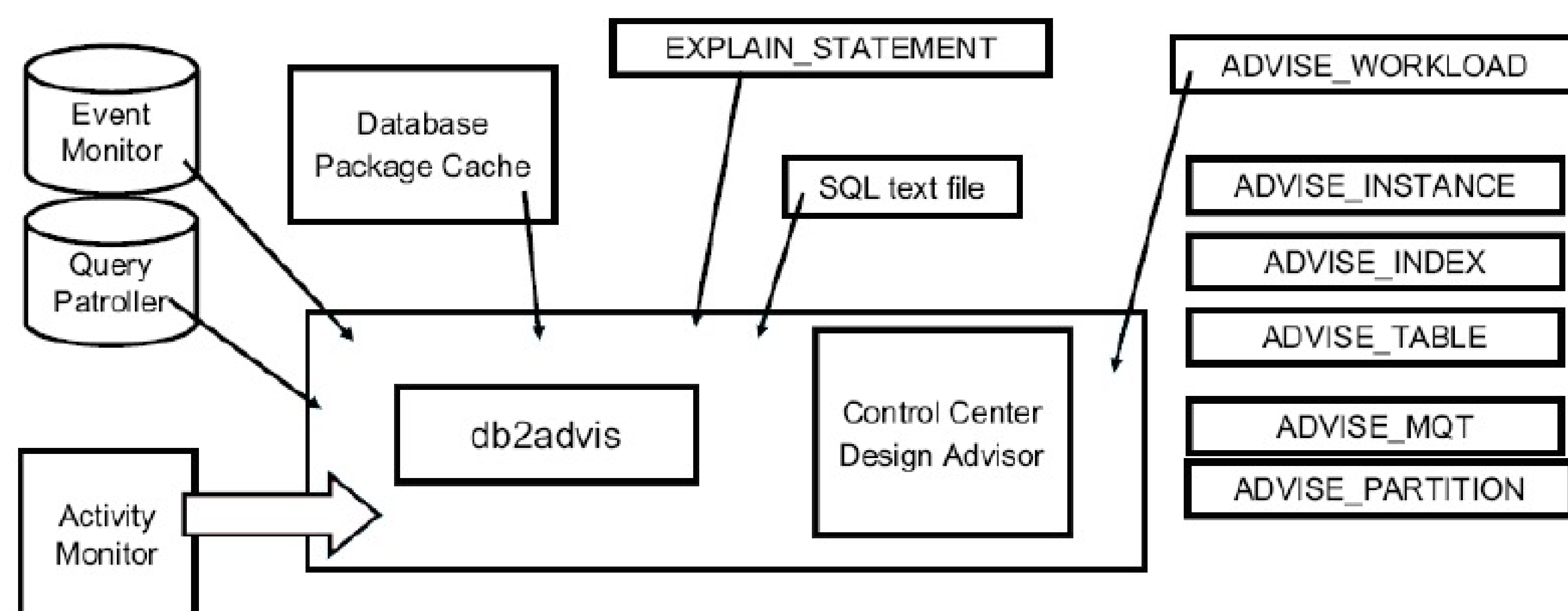


图 6-10 Design Advisor

Design Advisor 提供的建议能与数据库调优专家的建议相媲美。对于非专家来说，该工具的好处是可以获得更好的设计。对于专家来说，Design Advisor 可以节省他们宝贵的时间，因为 Design Advisor 可以提供初始设计，然后由专家进一步改进设计。Design Advisor 还可以提供对专家设计的独立确认。

要使用 Design Advisor，首先需要在相应数据库里执行 EXPLAIN.DDL，创建并存储各种数据：

```
cd $INSTHOME/sqllib/misc
db2 -tvf EXPLAIN.DDL
```

使用 Design Advisor 的第一步是收集和描述提供给 Design Advisor 的工作负载。可以让 Design Advisor 从以下输入获取工作负载：

- 最近的 SQL 语句(来自动态 SQL 快照)
- Query Patroller 语句(更适用于数据仓库数据库)
- 静态 SQL 语句(来自应用程序包)
- 解释后的 SQL 语句
- Event Monitor 语句
- 包含事务的文件
- Activity Monitor 捕获的 SQL 语句

我们可以为每个事务(即每条 SQL 语句)赋予值为 1、10、100 或 1000 的频率。这将导致 Design Advisor 对频率值为 10 的事务的重视程度是对频率值为 1 的事务的重视程度的 10 倍。

收集完事务并设置好每个事务的频率后，就可以运行命令 db2advis 来获取关于索引的建议。如果创建索引，SQL 执行成本可以提高 32.42%。

```
    db2advis -d sample -i top.sql -disklimit 2 -o newindex.ddl > advis.out
    cat advis.out
    execution started at timestamp 2015-12-17-11.54.08.236000
    found [3] SQL statements from the input file
    Recommending indexes...
    total disk space needed for initial set [ 0.817] MB
    total disk space constrained to [ 2.000] MB
    Trying variations of the solution set.
    Optimization finished.
    1 indexes in current solution
    [1118.4197] timerons (without recommendations)
    [755.8014] timerons (with current solution)
    [32.42%] improvement
    -- LIST OF RECOMMENDED INDEXES
    -- ===========================
    -- index[1], 0.817MB
    CREATE INDEX "ADMIN"."IDX411171655470000" ON "ADMIN"."HISTORY"
("TELLER ID" ASC,"BRANCH ID" ASC) ALLOW REVERSE SCANS ;
    COMMIT WORK ;
    RUNSTATS ON TABLE "ADMIN"."HISTORY" FOR INDEX
"ADMIN"."IDX411171655470000" ;
```

```
COMMIT WORK ;
```

现在我们将讲解同一个案例研究。

下面是这个案例研究中使用到的命令：

```
db2advis -d sample -i top1.sql -m IMCP -k LOW  -l 700 -c TEMP_TBS -f
```

要点包括：

- -m IMCP：规定 Design Advisor 应该考虑新的索引(I)和 MQT(M)、将标准的表转换成 MDC 表(C)、重新为已有的表分区(P)。默认情况是只考虑索引。
- -k LOW：规定将工作负载压缩到 LOW 级别。这里 Design Advisor 将分析提供的更大一组的工作负载。默认情况是中等(MEDIUM)。
- -l 700：规定任何新的索引、MQT 等都不能消耗多于 700 MB 的空间。默认情况是数据库总体规模的 20%。
- -c TEMP_TBS：规定使用表空间 TEMP_TBS 作为生成 MQT 建议的临时工作空间。如果想要 MQT 建议，并且正在运行 DPF(多分区)示例，那么这个选项是必需的。否则，这个参数是可选的。
- 另一个有用的选项(没有给出)是-o *output_file*。该选项保存脚本，以便在文件中创建建议的对象。

当该命令执行时，它描述正在进行的工作，下面显示了其中的一部分。至此，Design Advisor 已经生成了有关除 MDC 之外的所有对象的建议。

```
Cost of workload with all recommendations included [1306186] timerons
27 indexes in current solution
3 partitionings in current solution
8 MQTs in current solution
```

建议集有 27 个索引(新索引或已有的索引)、3 个分区(与 DPF 相关的更改，例如新的分区键或表空间)以及 8 个 MQT(新的或已有的)。

接下来，Design Advisor 分析 MDC，并在完成时显示以下信息：

```
3 clustering dimensions in current solution
[12305400] timerons (without any recommendations)
[1042873] timerons (with current solution)
[91.53%] improvement
```

“3 clustering dimensions”意味着 Design Advisor 建议 3 个 MDC 维。这 3 个维可以同时在一个表上，也可以在不同的表上。例如，3 个维都在表 A 上，或者其中 1 个维在表 A 上，另外两个维在表 B 上。性能统计信息指的是所有建议的性能，而不仅仅是 MDC

建议的性能。“timerons (without any recommendations)”指的是现有设计消耗的时间(注意 timerons 不是秒)，而“timerons (with current solution)”指的是实施这些建议后估计消耗的时间。

接着，Design Advisor 以 DDL 格式显示建议，并且该 DDL 已经被注释掉。这些建议以如下顺序出现：

(1) 包括 MDC 或分区建议的基本表。

(2) MQT 建议(首先是新的 MQT，然后是要保留的已有的 MQT，最后是未使用的 MQT)。

(3) 新的集群索引(如果有的话)。

(4) 索引建议(新的，保留的，然后是未使用的)。

关于更改表的建议如下：

```
-- CREATE TABLE "ORACLE"."LINEITEM" ("L ORDERKEY BIGINT NOT NULL,
--         "L PART" INTEGER NOT NULL,
--         "L_SUPPKEY" INTEGER NOT NULL,
--         "L LINENUMBER" INTEGER NOT NULL,
--         "L SHIPINSTRUCT" CHAR(25)  NOT NULL,
(11 other columns omitted from this example)
--       MDC409022109290000 GENERATED ALWAYS AS ( ((INT(L_SHIPDATE))/7) )
--       PARTITIONING KEY ("L PARTKEY") USING HASHING
--       IN "TPCDLADT"
-- ORGANIZE BY (
--       MDC409022109290000,
--       L SHIPINSTRUCT )
-- PARTITIONING KEY (L ORDERKEY) USING HASHING
-- IN TPCDLDAT
--;
-- COMMIT WORK ;
```

注意：

这里建议使用新的分区键(L_ORDERKEY)替代当前的分区键(L_PARTKEY)，后者被注释掉了。这个表的 MDC 建议(ORGANIZE BY 子句)包括两个维：一个生成的列((INT(L_SHIPDATE)/7))和一个已有的列(L_SHIPINSTRUCT)。

输出中接下来的是关于 MQT 的建议，如下所示：

```
-- LIST OF RECOMMENDED MQTs
-- ========================
-- MQT MQT40902204140000 can be created as a refresh immediate MQT
```

```
-- mqt[1],   0.009MB
CREATE SUMMARY TABLE "ADVDEMO2"." MQT40902204140000"
   AS (SELECT Q6.CO AS "CO", Q6.C1 AS "C1", …additional details omitted
here…)
   DATA INITIALLY DEFERRED REFRESH IMMEDIATE PARTITIONING KEY (C8)
   USING HASHING IN TPCDLDAT ;
COMMIT WORK;
REFRESH TABLE "ADVDEMO2"." MQT40902204140000";
COMMIT WORK;
RUNSTATS ON TABLE "ADVDEMO2"." MQT40902204140000";
COMMIT WORK;
-- MQT MQT409022041530000 can be created as a refresh immediate MQT
(… DDL to create this table follows…)
```

MQT 建议包括：估计的大小、使用的表空间、分区键(如果适用的话)、刷新类型(立即或延迟)，以及这个表是否是基本表的复制品(由 REPLICATE 关键字表明，在本案例中不是)。

最后，Design Advisor 以下面显示的信息结束：

```
8604 solutions were evaluated by the advisor
DB2 Workload Performance Advisor tool is finished.
```

对于经常使用的查询以及大型或复杂的查询，将这些 SQL 语句作为输入使用 Design Advisor 来建议索引。

最后有两点需要注意的地方：确保在 Design Advisor 执行之前运行了 runstats；Design Advisor 给出的建议中会包含一项“UNUSED EXISTING INDEXES”，然后建议你删除一些索引。其实这里 Design Advisor 只是根据你提供的 SQL 进行分析，只要不能优化这个 SQL 的所有其他索引，它都认为是无用的，所以分析 Design Advisor 的结果时，也不要一味相信它，毕竟它不是真的专家。

6.9 本章小结

索引设计总结

创建的索引应该取决于表中数据的特点和表上的查询。

以下是创建不同索引时需要考虑的要点：

- 要避免某些排序，只要有可能，就通过使用 CREATE UNIQUE INDEX 语句定义主键和唯一键。

- 要改善数据检索，将 INCLUDE 列添加至唯一索引。合适的列为：不需要排序并且频繁被存取。
- 要加速多表关联的速度，考虑用连接列作为索引键。
- 要有效地搜索，决定对索引键使用升序还是降序，这取决于最常使用的次序。尽管当在 CREATE INDEX 语句中指定了 ALLOW　REVERSE　SCANS　参数时可以逆向搜索值，但是按指定索引次序的扫描比执行逆向扫描稍微快一些。
- 要节省索引维护成本和空间：

  ——尽量保证索引键之间不重复，例如已经有索引 ind1(a,b)，就不要再创建索引 ind2(a)。

  ——把不必要的字段也加入键不仅浪费空间而且降低查询效率。

  ——请参考以下实践准则

    - 对于在线事务处理(OLTP)环境，每个表创建 2 个或 3 个索引。
    - 对于只读查询环境，每个表可以创建 5 个以上索引。
    - 对于混合查询和在线事务处理环境，每个表可以创建 2 到 5 个索引。
- 要改进对父表执行的删除和更新操作的性能，请在外键上创建索引。
- 对于快速排序操作，在频繁用于排序数据的列上创建索引。
- 要想避免分页，又想减少表重组的次数，请定义集群索引。当定义表时，使用 PCTREE 关键字来指定页上应该留下多少可用空间，才能允许将插入行适当地放在页上。也可以指定 LOAD 命令的 pagefreespace 子句。
- 要启用联机索引整理碎片，创建索引时使用 MINPCTUSED 选项。MINPCTUSED 指定索引叶子页中最小使用空间量的阀值并启用联机索引整理碎片。这可以在键删除期间以性能损失为代价而减少重组的需要。

在下列情况下，考虑创建索引：

- 在最频繁处理的查询和事务的 WHERE 子句中所使用的那些列上创建索引。例如以下 WHERE 子句：

  ```
  WHERE  WORKDEPT=A01 OR WORKDEPT=E21
  ```

  通常将会从 WOPKDEPT 上的索引获益，除非 WORKDEPT 列包含许多重复值。
- 按查询需要的顺序在对行排序的一列或多列上创建索引。不仅在 ORDER BY 子句中，而且其他功能，如 DISTINCT 和 GROUP BY 子句也都需要排序。

  例如，以下示例使用 DISTINCT 子句：

  ```
  SELECT DISTINCT WORKDEPT
       FROM EMPLOYEE
  ```

数据库管理器可使用 WORKDEPT 上定义为升序或降序的索引来消除重复值。此时同一个索引也可用于 GROUP BY 子句中，以将值分组，如以下示例所示：

```
    SELECT WORKDEPT, AVERAGE(SALARY)
FROM EMPLOYEE
GROUP BY WORKDEPT
```

● 使用复合键创建索引，该键包含每个列。当用此方式指定索引时，会走完全索引扫描，这比读表更有效。
例如，考虑下列 SQL 语句：

```
SELECT LASTNAME
FROM EMPLOYEE
WHERE WORKDEPT IN ('A00', 'D11', 'D21')
```

如果为 EMPLOYEE 表的 WORKDEPT 和 LASTNAME 列定义索引，那么通过扫描索引而不扫描整个表可能会更有效地处理该语句。注意，因为该谓词基于 WORKDEPT，因此此列应是该索引的第一列。如果只是针对这条 SQL，使用 INCLUDE 列创建索引会更合适。可将唯一索引定义为：

```
CREATE UNIQUE INDEX x ON employee(workdept) INCLUDE (lastname)
```

指定 lastname 为 INCLUDE 列而不是索引键的一部分，意味着 lastame 不参与排序并且只储存在索引的叶子页上。

索引性能总结

考虑关于使用和管理索引的下列建议：

在创建索引时指定并行性

当在 SMP 机器主管的大型表上创建索引时，可考虑将 intra_parallel 设置为 YES(1)或 SYSTEM(－1)以利于改进并行性能。

可使用多个处理器来扫描数据并对数据排序。

为索引指定独立表空间

可将索引存储在与表数据不同的表空间中。这样，通过减少索引存储期间读或写磁头的移动，可以更有效地使用磁盘存储器。也可以在更快的物理设备上创建索引表空间。而且，可以将索引表空间指定给不同的缓冲池，这可以将索引页较长时间保存在缓冲区内，因为它们不与表数据页竞争。

当不将索引放在单独的表空间时，数据页和索引页使用相同的数据块大小和预取量。

如果对索引使用不同的表空间，那么可为表空间的所有特征选择不同的值。因为索引通常比表小，并且分布在更少的容器中，因此索引一般只具有较小的数据块大小，如 8KB 和 16KB。当 SQL 优化器选择存取方案时，会考虑用于表空间的物理设备的速度。

确保索引集群度

如果 SQL 语句需要排序(如 ORDER BY、GROUP BY 和 DISTINCT)，那么即使索引可以满足排序，但在下列情况下，优化器可能不选择索引：

——索引集群程度较低。可检查 SYSCAT.INDEXES 的 CLUSTERRATIO 和 CLUSTERFACTOR 列。

——表很小，以致对表数据排序比扫描索引后再读表的成本更低。

——对存取该表有多个索引可供选择。

创建集群索引之后，以传统方式执行 REORG TABLE，这可创建组织完好的索引。要重新对表做集群，可以执行排序和 LOAD 来代替。通常，只能在一个索引上对表做集群。在构建集群索引之后构建附加(INCLUDE)索引。集群索引试图维护数据的特定次序，以改进 RUNSTATS 实用程序收集的 CLUSTERRATIO 或 CLUSTERACTOP 统计信息。

要帮助维护集群及比率，在装入或重组表之前，在改变表时设定适当的 PCTFREE。PCTFREE 指定每页上的可用空间提供用于插入空间，以便可以适当地对这些插入预留空间做集群。如果不为表指定 PCTFREE，那么重组会使数据充分使用每个数据页。

注意：

目前，更新操作不会维护集群。换言之，如果更新一条记录以便其键值在集群索引中更改，那么不会将该记录移动至新页来维护集群次序。要想维护集群，使用 DELETE+INSERT 来替换 UPDATE。

保持表和索引统计信息为最新的

当创建新索引后，运行 RUNSTATS 实用程序来收集索引统计信息。这些统计信息使优化器知道使用索引是否可改进存取性能。

启用联机索引整理碎片

如果将 MINPCTUSED 子句设置为大于零，就对索引启用联机索引整理碎片。当页上的可用空间为指定级别或低于指定级别而索引仍可用时，联机索引整理碎片允许通过合并叶子页来压缩索引。

必要时重组索引

要从索引获取最佳性能，考虑定期重组索引，因为对表进行更新会使得索引页预取效

率降低。

要重组索引，删除并重新创建索引，或使用 REORG 实用程序。要减少对重组的需要，当创建索引时，指定适当的 PCTFREE 来在创建的每个索引叶子页上保留一定百分比的可用空间。在将来的活动中，可将记录插入索引，而使索引页分割的可能性减小。而分割导致索引页既不是连续的也不是按排序的，从而导致降低索引页预取的效率。

注意：

在重组索引时，保留创建索引时指定的 PCTFREE。删除并重新创建或重组索引也会创建大致连续且顺序排列的新的一组页，并改进索引页预取。尽管时间和资源方面的成本更高，但 REORG TABLE 实用程序也确保数据页的集群。集群为存取大量数据页的索引扫描带来很大的好处。在对称多处理机(SMP)环境中，如果 intra_parallel 数据库管理器配置参数为 YES 或 ANY，那么“传统的”REORG TABLE 方式(该方式使用阴影表进行快速表重装)可以使用多个处理器来重新构建索引。

分析关于索引使用情况的 EXPLAIN 信息

定期在最频繁使用的查询上运行 EXPLAIN，并验证每个索引是否至少使用了一次。如果有一个索引未在任何查询中使用，考虑删除该索引。

EXPLAIN 信息也允许查看在大型表上的表扫描是否是作为嵌套循环连接的内部表来处理的。如果是这样，那么连接谓词列上的索引要么丢失，要么被认为应用于该连接谓词的效率不高。

对大小变化范围大的表使用易变(volatile)表

易变表是大小在运行时可以从空变为很大的一种表。对于这种表，其中的基数变化范围很大，优化器可能生成适合表扫描而不是索引扫描的存取方案。

使用 ALTER TABLE...VOLATILE 语句声明表是“易变的”，允许优化器对易变表使用索引扫描。在下列情况下，无论统计信息如何，优化器将使用索引扫描，而不是使用表扫描：

——引用的所有列都在索引中。

——索引可以在索引扫描中应用谓词。

技巧：

使用 db2advis 工具来为特定 SQL 语句创建一组最佳索引。

第 7 章

DB2 优化器

让我们以一条语句来开始我们的 DB2 优化器之旅，这条语句并不是以我们熟悉的SQL(如“select c1, c2 from …”)语句的方式给出的，而是以我们更加熟悉的陈述方式给出：“今天我需要去商店买冷饮，给汽车加油，还有寄包裹”。假如我们有一个机器人(名字叫DB2 优化器)负责决定如何完成这些任务，默认的，它可能会按照我们输入的指令的顺序来完成任务，这样做任务当然可以完成，但很可能不是完成任务的最好方式。设想以下情形：

- 如果先买冷饮，冷饮会不会在完成其他任务的过程中化掉?
- 如果在去加油站的路上油就用光了怎么办?
- 当我们到邮局的时候邮局会不会下班了？

是的，机器人 DB2 优化器需要做的是给我们提供完成这些任务的最佳方案，然而至少目前它无法很好地完成这件事情，为什么? 因为信息不足!

我们需要提供足够的信息，比如商店、邮局、加油站的位置；商店、加油站、邮局的营业时间；冷饮可以保留多久不会化掉；我们现在车里有多少油? 提供的信息越完善，越准确，我们越有可能得到最佳方案，否则，NO 信息，NO 优化！那么需要提供什么信息呢？要想 DB2 优化器给我们制定最优方案，需要提供如下信息：

- 准确实时的统计信息
- 合理配置的数据库配置参数
- 正确的优化级别
- 设计、创建合理的索引

- 磁盘 I/O 的合理设计(对应表空间的 TRANSRATE 和 OVERHEAD 参数)
- 高效优化的 SQL 语句

为了在数据库中执行查询或 DML 语句(INSERT、UPDATE、DELETE)，DB2 必须创建访问计划(ACCESS PLAN)。访问计划定义了按什么方式访问表，使用哪些索引，以及用何种连接(JOIN)方法来关联表等工作。好的访问计划对于 SQL 语句的快速执行至关重要。访问计划是由 SQL 编译器生成的，而优化器则是编译过程最核心、最重要的组成部分，因此有时用 DB2 优化器指代整个 SQL 编译器。DB2 优化器是一种基于成本的优化器，这意味着它根据表和索引的相关统计信息来估算不同访问计划的成本，进而选择最优的访问计划。

本章主要讲解如下内容：

- DB2 编译器介绍
- SQL 语句编译过程
- 优化器组件和工作原理
- 扫描方式
- 连接方法
- 优化器级别
- 基于规则的优化
- 如何影响优化器来提高性能

7.1 DB2 编译器介绍

SQL 编译器是 DB2 的心脏和灵魂(可以把它类比成宝马 730 或波音 747 的发动机引擎)。它分析 SQL 语句并确定可以满足每条语句的最有效的存取路径(请参阅图 7-1 和图 7-2)。SQL 编译器通过解析 SQL 语句来确定必须访问哪些表和列，从而明确所要完成的操作。DB2 优化器查询存储在 DB2 系统目录中的系统编目信息和统计信息，以确定完成满足 SQL 请求的任务的最佳方法。

在数据库中优化数据访问是 DB2 最强大的能力之一(在所有的数据库，包括 Oracle、DB2、Informix、Sybase 和 SQL Server 中，DB2 优化器是最强大的)。访问 DB2 数据时应告诉 DB2 要检索什么，而不是如何检索。无论数据实际上是如何存储和操作的，DB2 和 SQL 都可以访问数据。

SQL 语句的执行过程，可以想象成一个执行 4 步骤的过程：

(1) 接收并验证 SQL 语句的语法语义。

(2) 分析环境并优化满足 SQL 语句的方法。

(3) 创建计算机可读指令来执行优化的 SQL。

(4) 执行指令或存储它们以便将来执行。

这个过程的第(2)步是最关键的，这也是 DB2 优化器发挥作用的过程。优化器怎样决定如何以它的方式执行大量 SQL 语句？优化器有许多类型的优化 SQL 的策略，它如何选择使用这些策略中的哪一个？DB2 优化器是基于成本的优化器(Cost Based Optimizer)。这意味着优化器将始终尝试为每个查询制定减少总体成本的存取路径。要实现这个目标，DB2 优化器会应用查询成本公式，对每条可能的存取路径的 4 个因素进行评估和权衡：CPU 成本、I/O 成本、DB2 系统目录中的统计信息和实际的 SQL 语句，然后对多个访问计划进行成本估算，最终选出最优的访问计划。

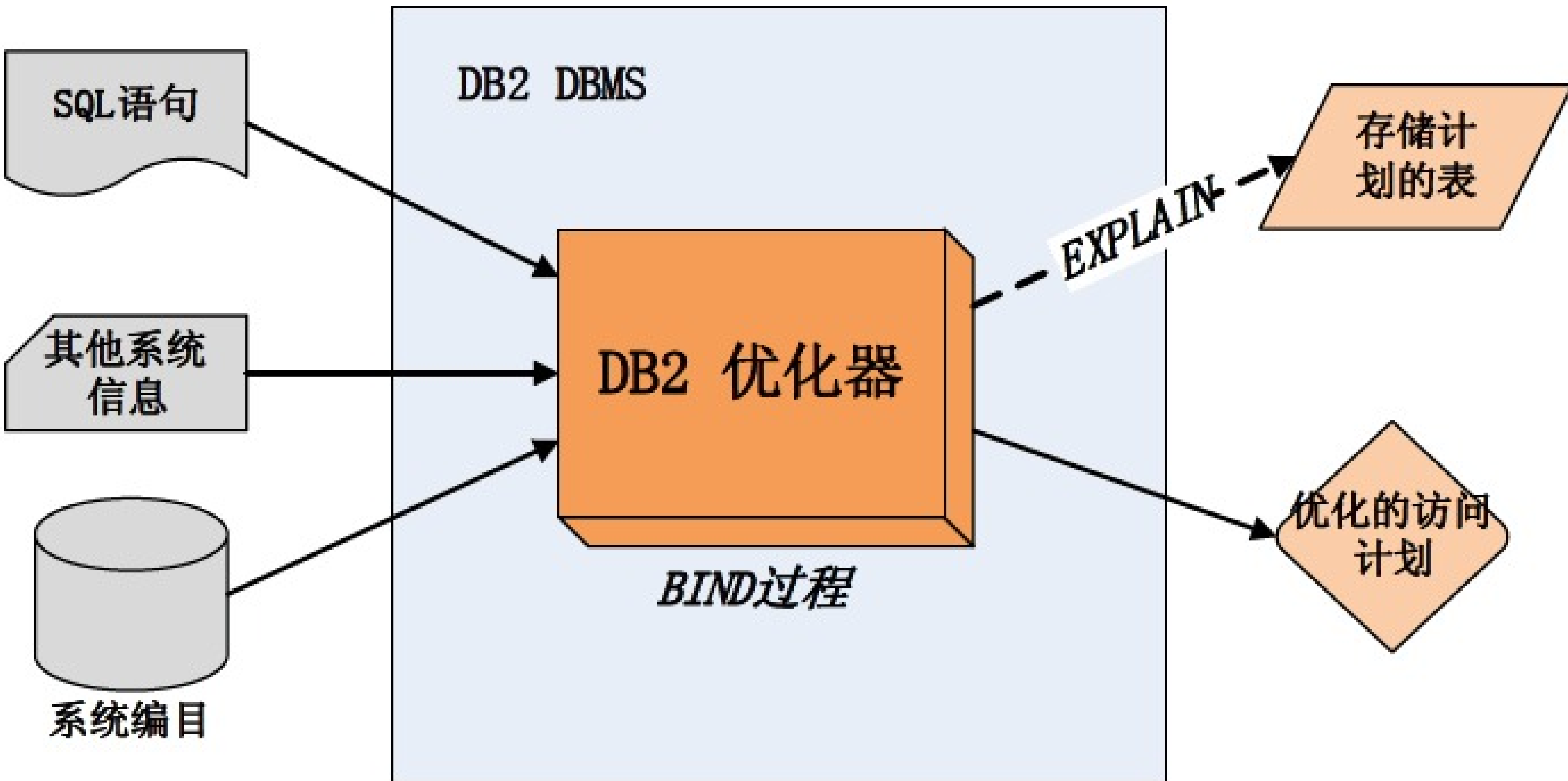


图 7-1　运行中的 DB2 优化器

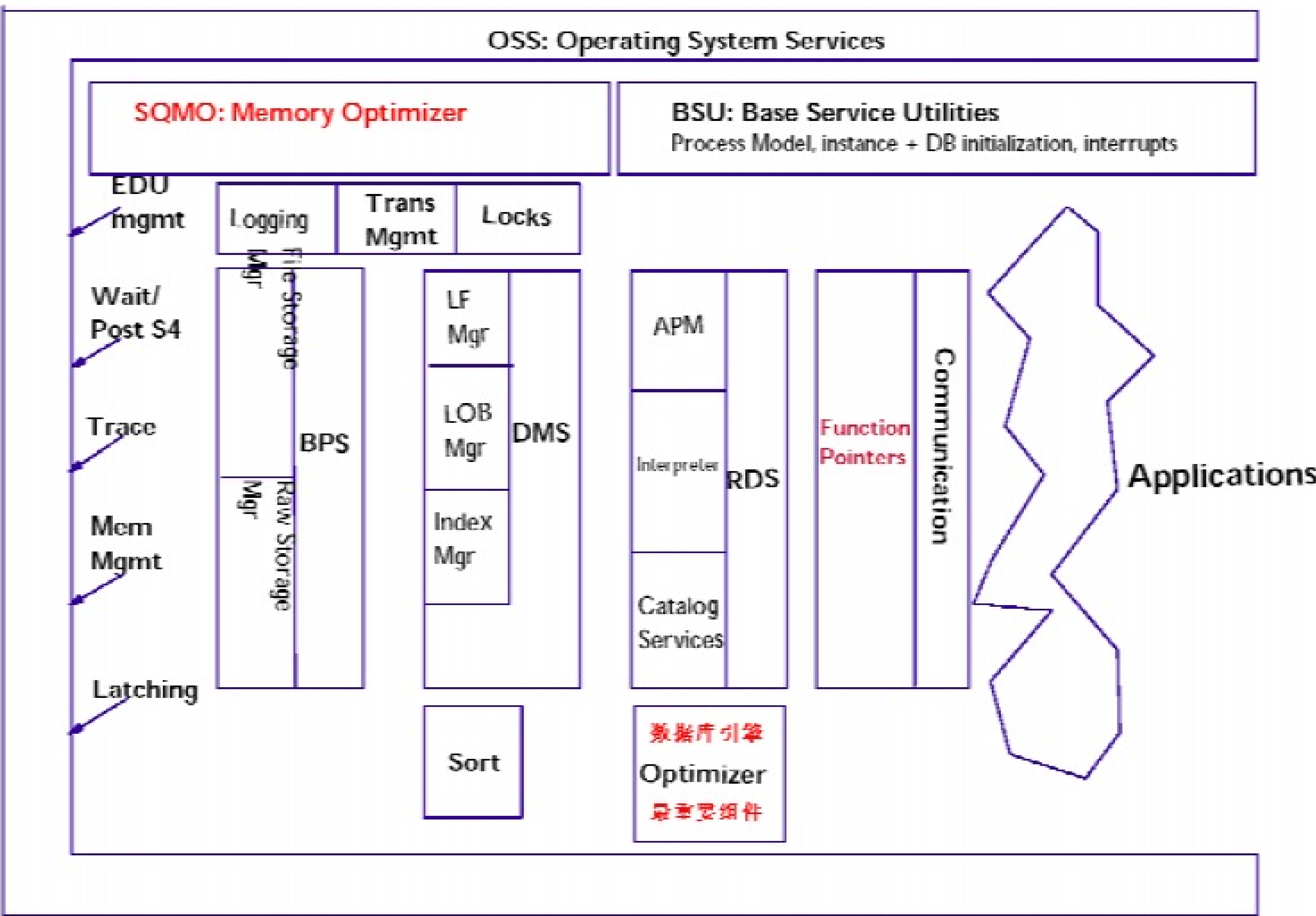


图 7-2　DB2 底层组件

7.2　SQL 语句编译过程

SQL 编译器执行几个步骤来产生可以执行的访问计划。这些步骤显示在图 7-3 中并在接下来进行了描述。注意，某些步骤仅针对联合数据库中的查询。

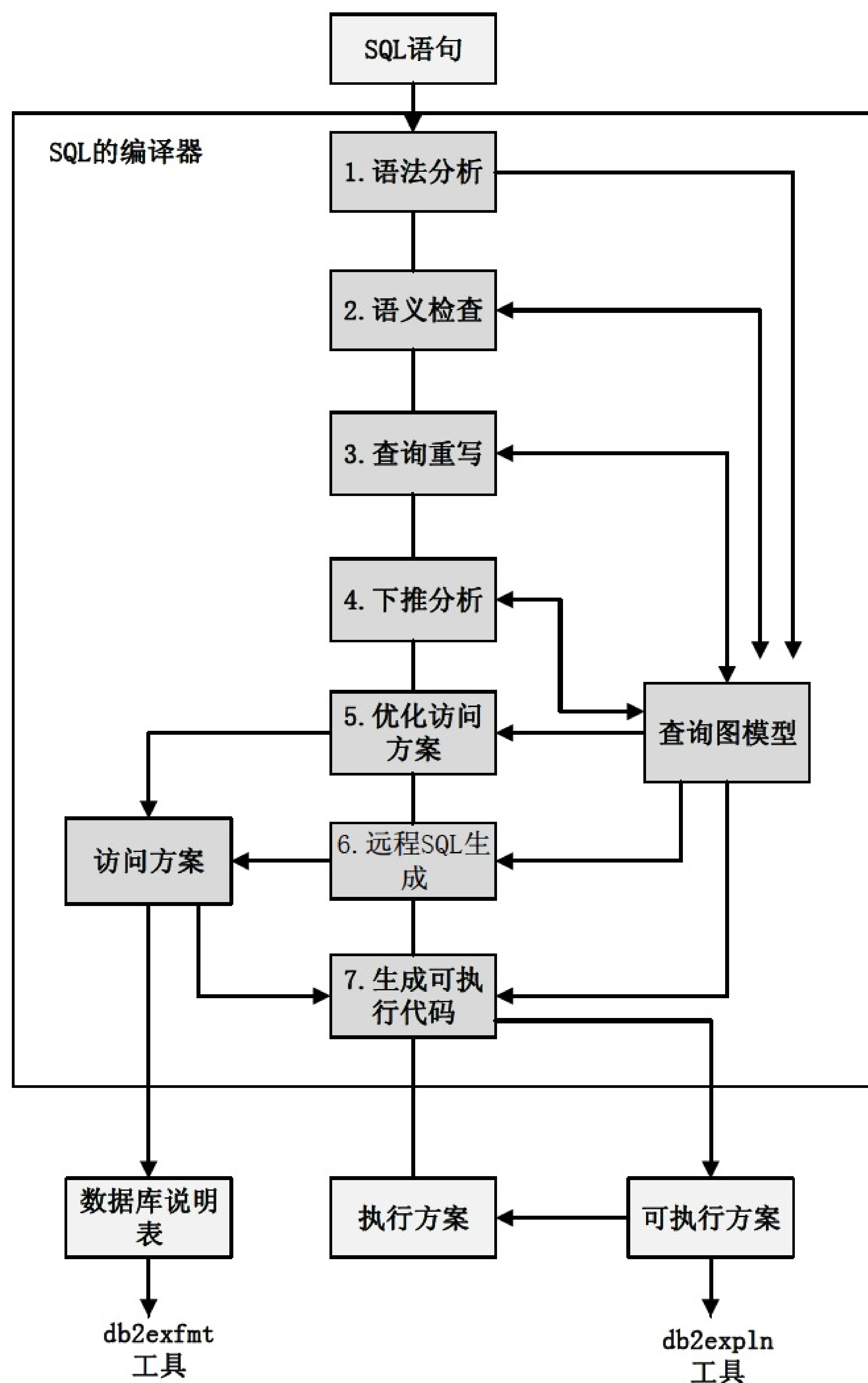


图 7-3　SQL 编译器执行的步骤

根据图 7-3 中描述的过程，SQL 语句的编译需要有如下流程：

1. 语法分析

SQL 编译器分析查询以验证其语法。如果检测到任何语法错误，例如“给汽车‘漏’油”中的语法错误，那么查询编译器停止处理，并将适当的错误代码返回至提交该查询的应用程序。当解析完成时，创建该查询的内部表现形式并将它存储在查询图模型中。查询图模型(Query Graph Model，QGM)是 DB2 编译器内部一种数据模型。

2. 语义检查

编译器确保在语句的各个部分中没有不一致。作为语义检查的简单示例，编译器验证为 YEAR 标量函数指定的列的数据类型是日期时间数据类型。

编译器也将行为语义添加至该查询图模型，包括参考约束、表检查约束、触发器和视图的作用。该查询图模型包含查询的所有语义、查询块、子查询、相关、派生的表、表达式、数据类型、数据类型转换、代码页转换和分区键。

3. 查询重写

编译器使用存储在查询图模型中的全局语义来将该查询变换为更易于优化的一种形式，并将结果存储在查询图模型中。例如，编译器可能会移动谓词，更改其应用级别，从而有可能提高查询性能。这种类型的操作移动称为一般谓词下推。在分区数据库环境中，下列查询操作的计算更为集中：

- 聚集
- 行的重新分布
- 相关子查询，这些子查询包含对该子查询外部的表列的引用

对于分区数据库环境中的某些查询而言，在重写查询的过程中，可能会发生解除相关(decorrelation)。在数据库分区之间移动结果集的成本很高，在查询重写过程中会尝试减小必须广播至其他数据库分区的内容的大小并且/或减少广播次数。

4. 下推分析

此步骤的主要任务是向优化器建议是否可在数据源处对操作以远程方式求值(即下推)。这种类型的下推活动仅适用于数据源查询，它是对一般谓词下推操作的扩展。除非执行联合数据库查询，否则绕过此步骤。

5. 优化访问方案

以查询图模型作为输入，编译器的优化器部分生成可以满足查询的多个备选执行方案。优化器综合使用有关表、索引、列和函数的统计信息来估算每个备选方案的执行成本，

然后选择具有最小估计执行成本的方案。优化器使用查询图模型来分析查询语义，并获取有关各种因素的信息，包括索引、基本表、派生的表、子查询、关联和递归。

优化器还可考虑另一类下推操作，聚集(AVG、SUM、MAX、MIN、COUNT 等)与排序，可将这些操作的求值下推到“数据管理服务”(DMS)组件中来提高性能。

在确定页大小选择时，优化器还考虑是否有不同大小的缓冲池，该环境是否包括分区数据库，以及是否能够为在对称多处理器(SMP)环境中实现分区内并行性改善选择的方案均属考虑范围。优化器使用此信息来帮助选择该查询的最佳访问方案。

编译器此步骤的输出是访问方案。此访问方案提供说明表中捕获的信息。可以用说明快照捕获用于生成该访问方案的信息。

6. 远程 SQL 生成

优化器选择的最终方案可由一组对远程数据源执行操作的步骤组成。对于每个数据源执行的那些操作。这个步骤仅限于联合数据库。

7. 生成可执行代码

在最后一个步骤中，编译器使用访问方案和查询图模型来创建查询的可执行访问方案或节(SECTION)。此代码生成步骤使用查询图模型中的信息，以避免重复执行对一个查询只计算一次的表达式。这种优化还适用于代码页转换和绑定(宿主)变量。

要对具有绑定(宿主)变量、专用寄存器或参数标记的静态和动态 SQL 语句启用查询(重新)优化，应在绑定程序包时使用 REOPT 绑定选项。如果使用此选项，将使用绑定(宿主)变量、参数标记或专用寄存器的值而不是由编译器选择的默认估计值，来优化属于该程序包并且包含这些变量的 SQL 语句的访问路径。当值可用时，执行查询时就会进行此优化。

有关静态 SQL 语句访问方案的信息存储在系统目录表(SYSCAT.STATEMENTS)中。当执行该程序包时，数据库管理器将使用存储在系统目录表中的信息来确定如何访问该数据，并提供该查询的结果。此信息也可以由 db2expln 解释工具使用。

应当在经常更改的表上以适当的时间间隔执行 RUNSTATS。优化器需要有关表及其数据的最新统计信息以创建最有效的访问方案。可以通过重新绑定应用程序以利用更新的统计信息。如果未执行 RUNSTATS 或优化器怀疑在空表或几乎是空的表上执行了 RUNSTATS，那么优化器可能使用默认值或尝试根据在磁盘上存储该表所用的文件页的数目(FPAGES)来得出特定的统计信息。

7.3　优化器组件和工作原理

理解 DB2 优化器的工作机制对于性能调优是非常重要的。DB2 优化器主要由以下 3 种组件构成：

1. Query Rewriter(查询重写)

SQL 是一种非常灵活的查询语言，通常会有很多不同的语句都是达到同一目的；有时，优化器通过查询重写(Query Rewriter)会将一种形式的语句转换成另一种形式，前提是第二种形式的语句执行起来更有效。

2. Estimator(成本评估器)

成本评估产生 3 种度量标准：

- SELECTIVITY：表示有多少 ROWS 可以通过谓词被选择出来，大小介于 0.0~1.0，0 表示没有 ROW 被选择出来。如果没有收集统计信息(STATISTICS)，成本评估器(estimator)会使用默认的 SELECTIVITY 值，这个值根据谓词的不同而异。比如 '=' 的 SELECTIVITY 小于 '<'。如果有统计信息(STATISTICS)，比如对于“last_name = 'Smith'”，成本评估器使用“last_name”列的 DISTINCT 值的倒数(是指表中所有“last_name”的 DISTINCT 值)作为 SELECTIVITY。
- CARDINALITY：表示行集(ROW SET)的行数。
- COST：COST 表现了 DISK I/O、CPU USAGE 资源单位的使用成本单位(timeron)。

3. Plan Generator(计划生成器)

计划生成器的作用是尝试各种可能的执行计划，选择成本最低的一种。

7.3.1　查询重写示例：谓词移动、合并和转换

在查询重写阶段，查询编译器将 SQL 语句变换为更易于优化的格式，并因此改善可能的访问路径。查询重写对于非常复杂的查询(包括具有许多子查询或许多连接的查询)特别重要。查询生成器工具通常会创建这些很复杂的查询。

可更改优化级别来影响应用于 SQL 语句的查询重写规则的数目。要查看查询重写的一些结果，可以使用 db2expln、dynexpln、db2exfmt 等工具。

可以使用下列 3 种主要方式中的任何一种查询重写方式：

1. 谓词合并

要构造查询以便它具有最少的操作数(尤其是 SELECT 操作)，SQL 编译器查询重写来合并查询操作。以下是谓词合并的一些操作：

- 视图合并：使用视图的 SELECT 语句可限制表的连接顺序并可引入表的冗余连接。如果在查询重写期间合并视图，那么可撤销这些限制。
- 子查询至连接的变换：如果某条 SELECT 语句包含子查询，那么可以限制表的顺序处理的选择。
- 消除冗余连接：在查询重写期间，可除去冗余连接来简化 SELECT 语句。
- 共享聚集(AVG、MAX、SUM、COUNT 和 MIN)：当查询使用不同的函数时，查询重写可减少需要执行的计算数。

2. 谓词移动

要构造具有最少数目的操作和谓词的查询，编译器查询重写以移动查询操作。以下是谓词移动的一些操作：

- 消除 DISTINCT：在查询重写期间，优化器可移动执行 DISTINCT 操作的位置，从而减少此操作的成本。在提供的扩展示例中，全部除去了 DISTINCT 操作。
- 一般谓词下推：在查询重写期间，优化器可更改应用谓词的顺序，以便将选择性更强的谓词尽早地应用于该查询。
- 解除相关：在分区数据库环境中，在数据库分区之间移动结果集的成本很高。减小必须广播至其他数据库分区的信息的大小和/或减少广播次数，是查询重写的目标。

3. 谓词转换

SQL 编译器查询重写，以便把特定查询的现有谓词转换为更优的谓词。以下是可进行谓词转换的一些操作：

- 添加隐含谓词：在查询重写期间，可以将谓词添加至查询，以允许优化器在选择该查询的最佳访问方案时考虑其他表连接。
- OR 至 IN 的变换：在查询重写期间，可以将 OR 谓词转换为 IN 谓词，以获取更有效的访问方案。SQL 编译器也可将 IN 谓词转换为 OR 谓词，但前提是此变换将创建一种更有效的访问方案。

查询重写示例：视图合并

假定要访问 EMPLOYEE 表的以下两个视图：一个视图显示受过高等教育的职员，另

一个视图显示工资超过 35 000 美元的职员。这两个视图的定义如下所示：

```
    CREATE VIEW EMP EDUCATION (EMPNO, FIRSTNME, LASTNAME, EDLEVEL) AS
SELECT EMPNO, FIRSTNME, LASTNAME, EDLEVEL   FROM EMPLOYEE        WHERE EDLEVEL
> 17
    ----------------------------------------------------------------------
    CREATE VIEW EMP SALARIES (EMPNO, FIRSTNAME, LASTNAME, SALARY) AS      SELECT
EMPNO, FIRSTNME, LASTNAME, SALARY   FROM EMPLOYEE       WHERE SALARY > 35000
```

现在，假定执行下面的语句，查询既受过高等教育且工资又超过 35 000 美元的职员：

```
    SELECT E1.EMPNO, E1.FIRSTNME, E1.LASTNAME, E1.EDLEVEL, E2.SALARY
FROM EMP_EDUCATION E1,  EMP_SALARIES  E2    WHERE E1.EMPNO = E2.EMPNO
```

在优化器查询重写期间，可合并这两个视图来创建下面这个查询：

```
    SELECT E1.EMPNO, E1.FIRSTNME, E1.LASTNAME, E1.EDLEVEL, E2.SALARY     FROM
EMPLOYEE E1, EMPLOYEE E2        WHERE E1.EMPNO = E2.EMPNO          AND E1.EDLEVEL
> 17   AND E2.SALARY  > 35000
```

通过使用用户编写的 SELECT 语句来合并这两个视图中的 SELECT 语句，优化器在选择访问方案时可考虑更多选项。此外，如果已合并的这两个视图使用相同的基本表，还可执行其他重写。

4. 查询重写示例：子查询至连接的变换

SQL 编译器将执行包含子查询的查询，例如下面这条 SQL 语句：

```
    SELECT EMPNO, FIRSTNME, LASTNAME, PHONENO        FROM EMPLOYEE      WHERE
WORKDEPT IN                 (SELECT DEPTNO                        FROM DEPARTMENT
WHERE DEPTNAME = 'OPERATIONS')
```

将之转换为如下形式的连接查询：

```
    SELECT DISTINCT EMPNO, FIRSTNME, LASTNAME, PHONENO         FROM EMPLOYEE EMP,
DEPARTMENT DEPT  WHERE EMP.WORKDEPT = DEPT.DEPTNO           AND DEPT.DEPTNAME =
'OPERATIONS'
```

一般情况下，连接是基于集合操作的，而子查询是基于行集(rows set)操作的，所以连接的执行效率比子查询高很多。

查询重写示例：消除冗余连接

有时编写或生成的查询可能含有不需要的连接。在查询重写期间，也可能产生类似下

面的查询：

```
    SELECT E1.EMPNO, E1.FIRSTNME, E1.LASTNAME, E1.EDLEVEL, E2.SALARY
            FROM EMPLOYEE E1,  EMPLOYEE E2      WHERE E1.EMPNO = E2.EMPNO
AND E1.EDLEVEL > 17           AND E2.SALARY  > 35000
```

在此查询中，SQL 编译器可消除连接并将查询简化成如下形式：

```
    SELECT EMPNO, FIRSTNME, LASTNAME, EDLEVEL, SALARY          FROM EMPLOYEE
WHERE EDLEVEL > 17           AND SALARY  > 35000
```

另一个示例假设具有该部门号的 EMPLOYEE 和 DEPARTMENT 样本表之间存在参考约束。首先，创建一个视图。该视图定义如下：

```
    CREATE VIEW PEPLVIEW  AS SELECT FIRSTNME, LASTNAME, SALARY, DEPTNO, EPTNAME,
MGRNO FROM EMPLOYEE E DEPARTMENT D   WHERE E.WORKDEPT = D.DEPTNO
```

于是，诸如以下内容的查询：

```
    SELECT LASTNAME, SALARY FROM PEPLVIEW
```

将变成如下所示：

```
    SELECT LASTNAME, SALARY  FROM EMPLOYEE  WHERE WORKDEPT NOT NULL
```

注意，在这种情况下，即使用户知道可重新编写查询，他们可能也无法这样做，因为他们对基础表没有访问权。他们可能只对以上显示的视图具有访问权。因此，必须在数据库管理器内执行这种类型的优化。

查询重写示例：共享聚集(AVG、MAX、SUM、COUNT 和 MIN)

在查询中使用多个函数可生成几次计算，这会占用时间。将要在查询中执行的计算数减少可改进方案。SQL 编译器执行使用多个函数的查询，例如下面这条 SQL 查询：

```
    SELECT SUM(SALARY+BONUS+COMM) AS OSUM, AVG(SALARY+BONUS+COMM) AS OAVG,
            COUNT(*) AS OCOUNT  FROM EMPLOYEE;
```

优化器用下列方式变换该查询如下：

```
    SELECT OSUM,OSUM/OCOUNT OCOUNT  FROM (SELECT SUM(SALARY+BONUS+COMM) AS
OSUM,  COUNT(*) AS OCOUNT  FROM EMPLOYEE) AS SHARED_AGG;
```

此重写将该查询从 2 次求和 2 次计数减少为 1 次求和 1 次计数。

查询器重写示例：消除 DISTINCT

如果 EMPNO 列被定义为 EMPLOYEE 表的主键，那么以下查询：

```
SELECT DISTINCT EMPNO, FIRSTNME, LASTNAME  FROM EMPLOYEE
```

将通过除去 DISTINCT 子句重写成如下形式：

```
SELECT EMPNO, FIRSTNME, LASTNAME  FROM EMPLOYEE
```

在以上示例中，由于选择了主键，因此 SQL 编译器知道返回的每一行已经是唯一的。在这种情况下，DISTINCT 关键字是多余的。如果未重写该查询，那么优化器将需要通过必要的处理(例如排序)来构建一种方案，从而确保这些列是相异的。

查询重写示例：谓词下推

改变谓词通常应用的级别可改善性能。例如，假定以下视图提供部门“D11”中所有职员的列表，该视图定义如下：

```
CREATE VIEW D11_EMPLOYEE  (EMPNO, FIRSTNME, LASTNAME, PHONENO, SALARY, BONUS,
COMM)  AS SELECT EMPNO, FIRSTNME, LASTNAME, PHONENO, SALARY, BONUS, COMM
FROM EMPLOYEE              WHERE WORKDEPT = 'D11'
```

并假定执行以下查询：

```
SELECT FIRSTNME, PHONENO FROM D11_EMPLOYEE  WHERE LASTNAME = 'BROWN'
```

在编译器的查询重写阶段将把谓词 LASTNAME='BROWN' 下推到视图 D11_EMPLOYEE 中。这使得可以更快并可能更有效地应用该谓词。在本例中可能执行的实际查询转变为：

```
SELECT FIRSTNME, PHONENO    FROM EMPLOYEE
          WHERE LASTNAME = 'BROWN'    AND WORKDEPT = 'D11'
```

谓词下推并不限于视图。可以下推谓词的其他情况包括 UNION、GROUP BY 和派生的表(嵌套表表达式或公共表表达式)。

查询器重写示例：隐含谓词

以下查询产生所属部门是向“E01”报告的经理的列表，以及那些经理所负责的项目的列表：

```
SELECT DEPT.DEPTNAME DEPT.MGRNO, EMP.LASTNAME, PROJ.PROJNAME  FROM
DEPARTMENT DEPT,    EMPLOYEE  EMP,      PROJECT   PROJ
WHERE DEPT.ADMRDEPT = 'E01'          AND DEPT.MGRNO = EMP.EMPNO
```

```
AND EMP.EMPNO  = PROJ.RESPEMP
```

优化器将该 SQL 查询重写并添加以下隐含谓词：

```
DEPT.MGRNO = PROJ.RESPEMP
```

作为此查询重写的结果，优化器在尝试选择该查询的最佳访问方案时可以考虑其他连接。

除以上谓词传递闭包外，查询重写还根据等式谓词隐含的传递性派生其他的本地谓词。例如，以下查询将列示部门号大于“E00”的部门和在这些部门工作的职员的名称：

```
SELECT EMPNO, LASTNAME, FIRSTNAME, DEPTNO, DEPTNAME
FROM   EMPLOYEE EMP,  DEPARTMENT DEPT
WHERE  EMP.WORKDEPT = DEPT.DEPTNO  AND DEPT.DEPTNO > 'E00'
```

对于此查询，该重写阶段添加以下隐含谓词：

```
EMP.WORKDEPT > 'E00'
```

作为此查询重写的结果，优化器会减少要连接的行数。

查询重写示例：OR 至 IN 的变换

假定 OR 子句根据同一列将两个或更多个简单的等式谓词相连，如下面的 SQL 查询：

```
SELECT *  FROM EMPLOYEE
WHERE DEPTNO = 'D11' OR DEPTNO = 'D21'   OR DEPTNO = 'E21'
```

如果 DEPTNO 列上没有索引，就将 OR 子句转换为以下 IN 谓词以允许更有效地处理该查询：

```
SELECT  *   FROM EMPLOYEE   WHERE DEPTNO IN ('D11', 'D21', 'E21')
```

在某些情况下，数据库管理器可以将 IN 谓词转换为一组 OR 子句，以便执行索引 OR(IndexORing)运算。

7.3.2 优化器成本评估

成本评估产生 3 种度量标准：

1. SELECTIVITY

表示有多少比例的 ROWS 可以通过谓词被选择出来，大小介于 0.0~1.0，0 表示没有 ROW 被选择出来。如果没有收集统计信息(STATISTICS)，成本评估器会使用默认的 SELECTIVITY 值，这个值根据谓词的不同而异。比如 '=' 的 SELECTIVITY 小于 '<'。如

果有统计信息(STATISTICS)，比如对于“last_name = 'Smith'”，成本评估器使用“last_name”列的 DISTINCT 值的倒数(是指表中所有“last_name”的 DISTINCT 值)作为 SELECTIVITY。

如果 last_name 列上有分布统计信息，就使用分布统计信息根据 last_name 值的分布情况产生的 SELECTIVITY 作为 SELECTIVITY。分布统计信息在当列有数据分布不均匀时可以大大帮助 CBO 产生好的 SELECTIVITY。

2. CARDINALITY

表示行集(rows set)的行数，分为以下几种：

- base cardinality：base table 的行数。如果表的统计信息收集过了，就直接使用统计信息。如果没有，就使用表 extents 的数量来估计。
- effective cardinality：有效行集，是指从基表中选择出来的行数。是 base cardinality 和表上所有谓词的组合 selectivity 的乘积。如果表上没有谓词，那么 effective cardinality=base cardinality。
- join cardinality：两表 join 后产生的行数。是两表 cardinality 的乘积(Cartesian)乘以 join 谓词的 selectivity。
- distinct cardinality：列上 distinct 值的行数。
- group cardinality：GROUP BY 操作之后 row set 的行数。由 grouping columns 的 distinct cardinality 和整个 row set 的行数决定。

3. COST

COST 表现了 DISK I/O、CPU USAGE 资源单位的使用成本单位(timeron)。扫描方式决定从基表(base table)中获得数据所需的资源单位的使用数量(units of work or resource used)。也就是说，扫描方式决定 COST 的值。扫描方式可以是 TABLE SCAN、INDEX SCAN 和 RID SCAN。

DB2 优化器凭借精确的基数成本评估值来准确计算出每个待定查询访问计划的成本。基数成本评估是这样一种过程：在应用了谓词或执行了聚集之后，优化器使用统计信息确定部分查询结果的大小。对于访问计划的每个操作符，优化器将估计操作符的基数输出。应用一个或更多谓词可以减少输出流基数。基数估计通过不同谓词的选择性经过计算得到最终估计的行数。

7.3.3 本地谓词基数(cardinality)估计

我们假设表 T1 有 X、Y 列，总行数为 20，并且表 T1 在 X 列上索引。T1 表的分布情况如下：

```
x y
0 a
1 b
3 c
9 d
12 e
19 b
19 d
20 d
30 e
31 a
32 c
39 d
42 e
43 a
44 b
45 d
47 e
50 a
55 b
60 c
```

我们已经执行下面的 RUNSTATS，收集了表 T1 的统计信息：

```
db2 runstats on table db2admin.t1 WITH DISTRIBUTION AND SAMPLED DETAILED
INDEXES ALL
```

现在我们希望执行下面的 SQL 语句：

```
SELECT * FROM T1 WHERE x BETWEEN 10 AND 50 AND y = 'a'
```

对谓词“BETWEEN 10 AND 50”，我们首先看 X<=50 的选择性。由于 50 处于第 18 个值，因此选择性为 18/20=0.9。

对于 X>=10，由于 10 不在表 T1 的数据中，而第 1 个小于 10 的数为 9，处在数据分布的第 4 个位置，因此选择性为(20－4)/20＝0.8。P1 谓词的整个选择性为 0.8×0.9=0.72。

我们的执行计划是先应用索引扫描，并应用“x BETWEEN 10 AND 50”谓词，由此产生的基数估计为 20×0.72=14。

对 Y = 'a'谓词，RUNSTATS 显示 Y 有 5 个不同的值，Y = 'a'谓词的选择性为 1/5=0.2。在应用完第 1 个谓词的基础上再应用第 2 个谓词产生的基数估计是 14×0.2=2.8。

对于整个查询，优化器产生的基数估计是 2.8，这与 3 行的实际输出结果接近。实际

输出结果如下所示：

```
$db2 "SELECT * FROM T1 WHERE x BETWEEN 10 AND 50 AND y = 'a'"
X          Y
----------- -
        31 a
        43 a
        50 a
3 条记录已选择。
```

下面是 db2exfmt 输出的执行计划：

```
           Rows
          RETURN
          (   1)
           Cost
            I/O
            |
            2.8     -注：2.8 为应用 y = 'a'和 x BETWEEN 10 AND 50 产生的基数估计
          FETCH
          (   2)
          7.59383
             1
        /----+---\
     14           20          -注：14 为应用 x BETWEEN 10 AND 50 产生的基数估计
    IXSCAN    TABLE: DB2ADMIN
    (   3)          T1
   0.0228255
       0
      |
      20
 INDEX: DB2ADMIN
     INX_X
```

7.3.4　连接基数(cardinality)估计

假设有表 T1、T2，数据分布如下：

```
T1          T2
X Y          Y
----------------
1 A          B
2 B          B
```

```
2 C          D
4 D          D
7 E          F
7 F          F
7 G          H
7 H          H
9 I          J
9 J          J
```

假设有以下查询：

```
SELECT * FROM T1, T2 WHERE T1.X = 7 AND T1.Y = T2.Y
```

这个查询有两个谓词：一个是本地谓词，应用于本地查询上的谓词我们称为本地谓词，第 2 个谓词是在 Y 列上做连接操作。

对于 T1.X = 7 谓词，由于 7 这个值是使用频率最高的值，因此它的使用频率会被分布统计信息收集到，计算得到选择性为 4/10=0.4。在上述例子中，如果没有收集 X 列的分布统计信息，那么 T1.X=7 的选择性是 1/DISTINCT CARD=1/5=0.2。所以如果某列上的数据分布不均匀，建议收集该列的分布统计信息。

在估计 T1.Y = T2.Y 谓词时假设：

- 所有 T2.Y 值都被包含在 T1.Y 中。
- 两个表中的列值是均匀分布的。

计算是首先检查每个表的 Y 值的唯一值个数，在我们的例子中，T1 表的 Y 列含有 10 个不同值，T2 表包含 5 个不同值。

连接谓词的选择性算法是：

```
Selectivity (T1.A = T2.A):= 1 / max(colcard(T1.A), colcard(T2.A))
```

在本例中：

```
Selectivity (T1.y = T2.y)=1/max(10,5)=0.1
```

因此整个谓词的基数估计是：

```
Result cardinality=Card(T1) * Card(T2) * sel(T1.x=7) * sel(T1.y=T2.y)
           =10 * 10 * 0.4 * 0.1
           = 4
```

这与 4 行的实际输出结果相同。实际输出结果如下所示：

```
$db2 "SELECT * FROM T1, T2 WHERE T1.x = 7 AND T1.y = T2.y"
```

```
X          Y Y
----------- - -
        7 F F
        7 F F
        7 H H
        7 H H
  4 条记录已选择。
```

下面是 db2exfmt 的执行计划输出：

```
..............省略...........
              4                  -注：4 是应用谓词 T2."Y" = T1."Y"估计的基数
           HSJOIN
           (   2)
           15.1688
              2
        /-----+-----\
     10              4           -注：4 是应用谓词 T1.X = 7 估计的基数
   TBSCAN          TBSCAN
   (   3)          (   4)
   7.58326         7.58475
      ..............省略...........
```

如果上面的假设不成立，会出现什么结果呢？我们假设 T1 表数据不变，T2 表发生变化，如下所示：

```
T1          T2
X Y          Y
----------------
1 A          A
2 B          A
2 C          A
4 D          A
7 E          A
7 F          F
7 G          M
7 H          N
9 I          N
9 J          N
```

这样就不满足刚才那两个假设：

- 首先 T1.Y 没有被完全包含在 T2.Y 中，只有 A 和 F 被包含。同样 T2.Y 也没有被完全包含在 T1.Y 中。
- T2.Y 的列数据分布不均匀，A 就占了 50%。

对于下面的查询：

```
SELECT * FROM T1, T2 WHERE T1.x = 1 AND T1.y = T2.y
```

对于谓词 T1.X = 1，由于 1 不是最常出现的值，没有被收集频率统计信息，因此直接使用 1 作为出现频率估计，选择性为 1/DISTINCT CARDINALITY =1/10=0.1。

对于 T1.Y = T2.Y，依然采用上面的方法计算得到选择性为 1/(MAX(10,4))=0.1。

整个查询估计的基数是：

```
Result cardinality
= Card(T1) * Card(T2) * sel(T1.x=1) * sel(T1.y=T2.y)
= 10 * 10 * 0.1 * 0.1
= 1
```

这与实际查询结果为 5 条严重不符合，错误率为 80%。该查询的输出结果如下所示：

```
$db2 "SELECT * FROM T1, T2 WHERE T1.x = 1 AND T1.y = T2.y"
X           Y Y
----------- - -
          1  A A
          1  A A
          1  A A
          1  A A
          1  A A
  5 条记录已选择。
```

在这种情况下，我们需要考虑 JOIN 完成之后的数据分布情况以纠正错误估计。如果优化器知道了表的数据分布信息，优化器就能正确估计基数。比如下面的 SQL 语句：

```
SELECT * FROM T1, T2 WHERE T1.y = T2.y
```

此查询的输出结果如下所示：

```
$db2 "SELECT * FROM T1, T2 WHERE T1.y = T2.y"
X           Y Y
----------- - -
          1  A A
          1  A A
          1  A A
```

```
         1   A A
         1   A A
         7   F F
  6 条记录已选择。
```

执行 select distinct x as COLVALUE,count(*) as VALCOUNT from (SELECT * FROM T1, T2 WHERE T1.y = T2.y)，这样就可以得到中间结果集 X 列的分布如下：

```
COLVALUE VALCOUNT
  1        5
  7        1
```

对于谓词(T1.x = 1) AND (T1.y = T2.y)，估计基数为：

```
Result cardinality
= Card(T1 join T2) * sel(T1.x = 1)
= 6 * 5/6= 5
```

对于谓词(T1.x =7) AND (T1.y = T2.y)，估计基数为：

```
Result cardinality:
= Card(T1 join T2) * sel(T1.x = 7)
= 6* 1/6=1
```

这两种与实际结果就完全一致了。

7.3.5　分布统计信息

在上面的例子中，当我们应用 T1.X=7 来选取数据时，优化器默认认为 T1.X 列的数据分布是均匀的，所以这种情况下优化器评估的基数(CARDINALITY)是 1/DISTINCT。但实际上 T1.X 列的数据分布是不均匀的，这种情况下优化器评估出来的基数估计就不准确了。当确定表中数据分布不均匀时，可以运行包含 WITH DISTRIBUTION 子句的 RUNSTATS 命令。系统目录表中的统计信息通常包含关于表中最高值和最低值的信息，而优化器假定数据值是在两个端点值之间均匀分布的。然而，如果数据值的分布彼此之间差异较大，或者群集在某些点上，或者碰到许多重复的数据值，那么优化器就无法选择最佳的访问路径，除非收集了分布统计信息。使用 WITH DISTRIBUTION 子句还可以帮助查询处理没有参数标记或主机变量的谓词，因为优化器仍然不知道运行时的值是有许多行，还是只有少数行。可以收集两种数据分布统计信息：

1. 频率统计信息

这些统计信息提供关于 num_freqvalues 数据库配置参数的值所指定级别的具有最高重复值的数目和次高重复值的数目的列和数据值的信息。默认值是 10，建议将这个值设置在 10~100 之间。如果将 num_freqvalues 设置为零，那么不保留任何频率值的统计信息。还可以将 num_freqvalues 设置为每个表、统计信息视图和特定列的 RUNSTATS 选项。

2. 分位数统计信息

这些统计信息提供关于与其他值相比如何分布数据值的信息。称为 K 分位数，这些统计信息表示值 V，至少 K 个值位于该值或该值以下。可以通过按升序排序值来计算 K 分位数。K 分位数值是从范围的低端起第 K 个位置中的值。

要指定应该将列数据值分组成的部分数，将 num_quantiles 数据库配置参数设置为 2~32 767 之间的某个值。默认值为 20，确保对任何相等或小于或大于谓词的优化器估计误差最大为正或负 2.5%，而对任何 BETWEEN 谓词的最大误差为正或负 5%。要禁用分位数统计信息收集，将 num_quantiles 设置为 0 或 1。可以对每个表或统计信息视图以及特定列设置 num_quantiles。

注意：

如果没有在 RUNSTATS 命令的列或表级别上指定 num_freqvalues 和 num_quantiles，那么 num_freqvalues 的值将从 num_freqvalues 数据库配置参数中获取，而 num_quantiles 的值将从 num_quantiles 数据库配置参数中获取。如果指定了较大的 num_freqvalues 和 num_quantiles 值，那么执行 runstats 时将需要更多 CPU 资源和内存，此时可以通过 stat_heap_sz 数据库配置参数来增加可用的内存量。

可以为单个列或一组列修改频率和分位数统计信息的精确度。提高分布统计信息的精确度将导致更大的 CPU 和内存消耗，并占用更多的系统目录表空间。对于这些分布统计信息，只考虑对拥有选择谓词的最重要的查询而言最为重要的列。

当出现下列任何一种条件时，RUNSTATS 将不收集分布统计信息：

- 当将 num_freqvalues 配置参数设置为 0，以及将 num_quantiles 数据库配置参数设置为 0 或 1 时。
- 当每个数据值是唯一的时候。
- 当该列是 LONG、LOB 或结构化列时。
- 如果列中只有一个非空值。
- 声明的临时表。

何时收集分布统计信息

要决定是否应创建和更新给定表或统计信息视图的分布统计信息，考虑以下两个因素：

(1) 应用程序是否使用静态或动态 SQL 和 XQUERY 语句。

分布统计信息对不使用宿主变量的动态查询和静态查询最有用。当使用具有宿主变量的查询时，优化器只能有限地利用分布统计信息。

(2) 列中的数据是否是均匀分布的。

如果表中至少有一列的数据分布非常“不均匀”，并且该列频繁出现在等式或范围谓词中。比如，在类似如下所示的子句中，考虑收集分布统计信息：

```
WHERE C1 = KEY;
WHERE C1 IN (KEY1, KEY2, KEY3);
WHERE (C1 = KEY1) OR (C1 = KEY2) OR (C1 = KEY3);
WHERE C1 <= KEY;
WHERE C1 BETWEEN KEY1 AND KEY2;
```

可能发生两种类型的不均匀数据分布(可能一起发生)：

- 可能按一个或多个子间隔集群数据，而不是均匀地在最高和最低数据值之间分布。考虑以下列，其中，数据在范围(5,10)内集群，该列的数据分布如下所示：

```
C1
0.0
5.1
6.3
7.1
8.2
8.4
8.5
9.1
93.6
100.0
```

分位数统计信息可以帮助优化器处理这种类型的数据不均匀分布。

要帮助确定是否不是均匀分布的列数据，执行如下查询：

```
    SELECT C1, COUNT(*) AS OCCURRENCES           FROM  T1      GROUP BY C1
ORDER BY OCCURRENCES DESC;
```

● 重复数据值可能经常出现。考虑使用下列频率分布数据的列：

数据值	频率
20	5
30	10
40	10
50	25
60	25
70	20
80	5

要帮助优化器处理重复值，可以创建分位数和高频值统计信息。

要指定哪个级别的统计精度

要确定存储分布统计信息使用的精度，指定数据库配置参数 num_quantiles 和 num_freqvalues。还可以指定这些参数作为收集表或列的统计信息时的 RUNSTATS 选项。这些值设置得越高，RUNSTATS 创建和更新分布统计信息时使用的精度越大。但是，精度越大，在 RUNSTATS 执行期间和在目录表中所需要的存储器中需要使用的资源就越多。

对于大多数数据库，为 num_freqvalues 数据库配置参数指定 10~100 之间的数。理论下，应创建高频值统计信息，以便其余值的频率可近似等于最高频值的频率，或者相比之下忽略不计。数据库管理器收集的数目可能低于此数，因为只对出现多次的数据值收集这些统计信息。如果需要只收集分位数统计信息，那么将 num_freqvalues 设置为 0。

要设置分位数的数目，指定 20~50 之间作为 num_quantiles 数据库配置参数的设置。确定分位数数目的经验方法为：

● 确定在估计任何范围查询的行数时可允许的最大错误百分比。
● 如果谓词是 BETWEEN，那么分位数的数目应近似为 100/P；如果谓词是任何其他类型的范围谓词(<、<=、> 或 >=)，那么分位数的数目应近似为 50/P。

例如，25 个分位数导致的最大估计错误对于 BETWEEN 谓词应为 4%，而对于“>”谓词则应为 2%。通常，至少指定 10 个分位数。对于极端不均匀的数据才需要 50 个以上的分位数。如果只需要高频值统计信息，那么将 num_quantiles 设置为 0。如果将此参数设置为“1”，那么因为值的整个范围适合分位数，因此不收集分位数统计信息。

收集特定列的分布统计信息

为了提高 RUNSTATS 和后续查询方案分析的效率，可以仅收集查询在 WHERE、GROUP BY 和类似子句中使用的列的分布统计信息。还可以收集关于列的列组的基数统计信息。优化器使用这种信息来在它为引用组中列的查询估计选择性时检测列相关。

RUNSTATS 仅收集执行该命令的数据库分区上的表的统计信息。将此数据库分区的 RUNSTATS 结果推广到其他数据库分区。如果执行 RUNSTATS 的数据库分区不包含表的一部分，那么将请求发送到数据库分区组中持有表的该部分的第 1 个数据库分区。

下面举一些收集特定列的分布统计信息的例子。

下列示例说明使用 RUNSTATS 来收集包含数据分布信息的系统目录表统计信息的不同方法。

收集表和索引上的数据库统计信息，包含分布统计信息：

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION AND INDEXES ALL
```

收集表上的数据库统计信息以及索引上的详细统计信息，包含分布统计信息：

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

总的来说，分布统计信息对处理不均匀的数据分布非常有好处，可以让优化器更好地选择最优的执行计划。

7.3.6　列组统计信息对基数的影响

列组(COLUMN GROUP)统计信息将获得一组列的不同值组合的数目。通常，DB2 优化器可用的基本统计信息不检测数据相关性。列组的使用将给多个谓词的联合选择提供更准确的基数估计。优化器凭借精确的基数估计(ESTIMATED CARDINALITY)值来准确计算出每个待定查询访问计划的成本。基数估计是这样一种过程：在应用了谓词或执行了聚集之后，优化器使用统计信息确定部分查询结果的大小。对于访问计划的每个操作符，优化器将估计该操作符的基数输出。应用一个或更多谓词可以减少输出的基数。

在计算谓词对于基数估计值的组合过滤效果时，通常会假设这些谓词彼此之间是独立的。但在统计方面，这些谓词可以彼此关联。单独地处理多个谓词通常会导致优化器低估基数值，而基数值的低估又会导致优化器选择次优的访问计划。

优化器会考虑使用多列统计信息来检测统计关联，并更加准确地估计多个谓词组合的过滤效果。下面讲述优化器如何利用多列统计信息来检测统计关联，并更加准确地估算多个等式谓词对于应用了至少两个本地 IN、OR 和等式谓词的 SQL 语句的组合过滤效果，以及它们对于应用了某种等级的 OR 谓词的 SQL 语句的过滤效果。利用 DB2 列组统计信息，优化器可以在多个谓词相关联时确定更好的查询访问计划，并提高查询性能。

1. 多个本地等式和本地 IN 谓词的统计关联

如果 SQL 语句的 WHERE 子句使用了多个谓词，如下所示：

```
    C1=? AND C2 IN ( ?, ?, ? )
```

并且收集了(C1, C2)的多列统计信息的话，那么优化器就会试着检测这些谓词间的统计关联以提高基数估计值。如下谓词除外：

- 带有 IN 或 OR 操作符的连接谓词
- 带有不等式、LIKE 或 IS NULL 操作符的本地谓词
- 带有子查询的谓词

C1=? 谓词就是本地等式谓词，本地等式谓词是应用于单个表的等式谓词，描述如下：

```
    COLUMN = literal
```

其中，literal 可以是以下任一内容：

- 常量值
- 参数标记或主变量
- 专用寄存器(例如 CURRENT DATE)

C2 IN (?, ?, ?)谓词则是本地 IN 谓词，本地 IN 谓词是应用于同一表格——与本地谓词应用的表格相同——的等式谓词，描述如下：

```
    COLUMN IN ( <VALUE LIST> )
```

其中，<VALUE LIST>是以逗号隔开的一个或多个上述(在本地等式谓词中)literal 的列表。

相当于 IN 谓词的 OR 谓词可以代替 IN 谓词在 SQL 语句中指定，而且优化器将会在说明统计关联时按相同的方式加以处理，也就是下面的谓词写法：

```
    COL IN  ( literal_1, literal_2, ..., literal_n )
```

相当于：

```
     COL=literal_1 OR COL=literal_2 OR ... OR COL=literal_n
```

下面的例子说明了优化器如何检测本地 IN、OR 和等式谓词间的关联：

```
    a) COL 1 IN ( <VALUE LIST> ) AND COL 2=literal AND COL 3=literal
b) (COL 1=literal 1 OR COL 1=literal 2 OR ... OR COL 1=liternal n) AND
COL 2=literal AND ... AND COL m=literal
c) COL 1 IN ( <VALUE LIST> ) AND COL 2 IN ( <VALUE LIST> ) AND ... AND COL m
IN ( <VALUE LIST> )
```

```
d) (COL 1=literal 1 OR COL 1=literal 2) AND (COL 2=literal 1 OR COL 2=literal 2)
AND ... AND (COL m=literal 1 OR COL M=literal 2)
e) COL 1 IN ( <VALUE LIST> ) AND ... And COL m IN ( <VALUE LIST> ) AND
COL_1_2=literal AND ... AND COL_1_k=literal
f) (COL 1=literal 1 OR COL 1=literal 2) AND COL 2=literal AND COL 3=literal
g) (COL 1=literal 1 OR COL 1=literal 2) AND (COL 2=literal 1 OR COL 2=literal 2)
AND COL_3=literal
```

下面这些谓词是优化器不会考虑为其检测统计关联的谓词的例子：

```
    a) (COL 1=literal AND COL 2=literal) OR (COL 1=literal AND COL 2=literal
AND COL_3=literal)
b)((COL 1=literal AND COL 2=literal) OR (COL 1=literal AND COL 2=literal)) AND
COL 3=literal
c)( COL_1 IN ( <VALUE LIST> ) OR (COL_2 IN ( <VALUE LIST> ) ) AND COL_3=literal
```

2. 列组统计信息更新示例

下面举几个列组统计信息更新示例。

C1 IN (<VALUE LIST>) AND C2 =liternal

考虑对 SAMPLE 数据库的 EMPLOYEE 表执行如下查询：

```
    SELECT FIRSTNME, LASTNAME, JOB, WORKDEPT, SALARY        FROM EMPLOYEE
WHERE JOB IN ('CLERK', 'SALESREP') AND     WORKDEPT = 'A00'      ORDER BY JOB,
SALARY
```

该查询从 EMPLOYEE 表返回 4 条记录，输出结果如下所示：

```
    FIRSTNME     LASTNAME         JOB      WORKDEPT  SALARY
    ------------ --------------- -------- -------- -----------
    GREG         ORLANDO          CLERK    A00       39250.00
    SEAN         O'CONNELL        CLERK    A00       49250.00
    DIAN         HEMMINGER        SALESREP A00       46500.00
    VINCENZO     LUCCHESSI        SALESREP A00       66500.00
    4 record(s)  selected.
```

SAMPLE 数据库在创建最初，还没有在表上收集统计信息。为了保证执行计划的准确性，需要更新表 EMPLOYEE 的统计信息。下面的 RUNSTATS 命令在 EMPLOYEE 表的每个列上收集了统计信息，包括分布统计信息和所有在 EMPLOYEE 表中定义的索引上的详细统计信息(如果存在的话)。该命令如下：

```
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE  WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

下面使用 dynexpln 来查看该 SQL 语句的执行计划，执行计划如下所示：

```
    $dynexpln -d sample -q "SELECT FIRSTNME, LASTNAME, JOB, WORKDEPT, SALARY
FROM EMPLOYEE       WHEAND     WORKDEPT = 'A00'       ORDER BY JOB, SALARY" -t
    Estimated Cost = 7.669430
    Estimated Cardinality = 1.190476
    Access Table Name = ORACLE.EMP1  ID = 3,12
    |  #Columns = 4
    |  Avoid Locking Committed Data
    |  Evaluate Block/Data Predicates Before Locking Committed Row
    |  Relation Scan
    ..............................略..........................
```

在上述访问计划输出中，估计基数为 1.190476，基数估计值 1.190476 与实际返回的 4 行结果不符。这是因为优化器假定两个谓词是独立的，因为相关的索引或列组统计信息不存在。我们可以使用 RUNSTATS 命令在(JOB,WORKDEPT)上收集列组统计信息，以此为优化器提供适当的信息来检测两个列之间的统计关联(如果存在的话)，命令如下：

```
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE ON ALL COLUMNS  AND COLUMNS
((JOB,WORKDEPT)) WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

重复上面的步骤，再次解释查询，然后生成查询访问计划，优化器就会计算出更好的基数估计值，因为它在两个列上收集了列组统计信息。具有更好的基数估计值的查询访问计划输出如下所示：

```
     $dynexpln -d sample -q "SELECT FIRSTNME, LASTNAME, JOB, WORKDEPT, SALARY
FROM EMPLOYEE       WHERE JOB IN ('CLERK', 'SALESREP') AND     WORKDEPT = 'A00'
ORDER BY JOB, SALARY" -t
    Estimated Cost = 118.487167
    Estimated Cardinality = 5.000000
    Access Table Name = ORACLE.EMPLOYEE  ID = 2,6
    |  #Columns = 4
    |  Volatile Cardinality
    |  Avoid Locking Committed Data
    |  Evaluate Block/Data Predicates Before Locking Committed Row
    |  Relation Scan
    ..........................略..............................................
```

由于列组统计信息是均匀分布的统计信息，而实际数据分布是不均匀的，因此基数估计值 5 比实际值 4 高了一些。可能已经注意到了，访问计划本身的评估成本(ESTIMATED COST)并未随着基数估计值的增大而显著改变。这是因为本例中描述的例子都很简单，数

据量非常小(只有 42 条记录)。如果语句涉及更大的表和两个或更多个表的连接的话，查询访问计划的评估成本就很可能会因为基数估计值的提高而显著改变。

C1 IN (<VALUE LIST>) AND C2 IN (<VALUE LIST>)

这个示例解释说明了在两个 IN 谓词上的列组统计信息的效果。考虑以下检索某一部门的经理和设计人员的奖金和薪水的查询。该 SQL 查询如下所示：

```
    SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS, SALARY        FROM EMPLOYEE
WHERE WORKDEPT IN ('D11','D21') AND          JOB IN ('MANAGER','DESIGNER')
ORDER BY WORKDEPT, SALARY
```

该查询从 EMPLOYEE 表返回 12 条记录，输出结果如下所示：

```
    FIRSTNME     LASTNAME      WORKDEPT JOB        BONUS     SALARY
    ------------ ------------- -------- ---------- --------- -----------
    MASATOSHI     YOSHIMURA    D11       DESIGNER  500.00    44680.00
    JENNIFER     LUTZ          D11       DESIGNER  600.00    49840.00
    JAMES        WALKER        D11       DESIGNER  400.00    50450.00
    MARILYN      SCOUTTEN      D11       DESIGNER  500.00    51340.00
    BRUCE        ADAMSON       D11       DESIGNER  500.00    55280.00
    DAVID        BROWN         D11       DESIGNER  600.00    57740.00
    ELIZABETH    PIANKA        D11       DESIGNER  400.00    62250.00
    KIYOSHI      YAMAMOTO      D11       DESIGNER  500.00    64680.00
    WILLIAM      JONES         D11       DESIGNER  400.00    68270.00
    REBA         JOHN          D11       DESIGNER  600.00    69840.00
    IRVING       STERN         D11       MANAGER   500.00    72250.00
    EVA          PULASKI       D21       MANAGER   700.00    96170.00
    12 record(s) selected.
```

首先，在没有获得(JOB,WORKDEPT)列组统计信息的情况下检查访问查询计划和基数估计值。可以按如下方式在 EMPLOYEE 表上执行另外一个 RUNSTATS 命令来完成该检查：

```
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE   WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

由于前面收集的统计信息被最新的 RUNSTATS 命令清除了，因此先前收集的列组统计信息不复存在。再次运行 DYNEXPLN 来查看该 SQL 语句的执行计划，以此来检查优化器估计的基数。查询访问计划如下所示：

```
    $dynexpln -d sample -q "SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS,
SALARY       FROM EMPLOYEE      WHERE WORKDEPT IN ('D11','D21') AND
JOB IN ('MANAGER','DESIGNER')      ORDER BY WORKDEPT, SALARY " -t
    Estimated Cost = 7.668932
```

```
    Estimated Cardinality = 7.285715
    Access Table Name = ORACLE.EMP1  ID = 3,12
    |  #Columns = 6
    |  Avoid Locking Committed Data
    |  Evaluate Block/Data Predicates Before Locking Committed Row
    |  Relation Scan
    |  |  Prefetch: Eligible
    |  Lock Intents
    |  |  Table: Intent Share
    ......................略...........................................
```

基数估计值 7.285715 与实际返回的 12 行结果不符。在(JOB,WORKDEPT)上收集列组统计信息会在计算两个 IN 谓词的组合过滤效果时为优化器提供必要的信息，用以说明统计关联。RUNSTATS 命令如下：

```
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE   ON ALL COLUMNS AND COLUMNS
((JOB,WORKDEPT))   WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

重复上面的步骤，再次解释查询，生成查询访问计划之后，优化器会计算出更好的、与实际结果很接近的基数估计值 11.200000。具有更精确的基数估计值的查询访问计划如下所示：

```
    $dynexpln -d sample -q "SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS,
SALARY       FROM EMPLOYEE      WHERE WORKDEPT IN ('D11','D21') AND
JOB IN ('MANAGER','DESIGNER')      ORDER BY WORKDEPT, SALARY " -t
    Estimated Cost = 117.296455
    Estimated Cardinality = 11.200000
    Access Table Name = ORACLE.EMPLOYEE  ID = 2,6
    |  #Columns = 6
    |  Volatile Cardinality
    |  Avoid Locking Committed Data
    ..............................略................................
```

C1 IN (<VALUE LIST>) AND C2 IN (<VALUE LIST>) AND C3=literal

在这个例子中，将向刚才那个查询中添加第 3 个谓词，确定有哪些职工得到了 500 美元的奖金。该 SQL 查询如下所示：

```
    SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS, SALARY  FROM EMPLOYEE
WHERE WORKDEPT IN ('D11','D21') AND        JOB IN ('MANAGER','DESIGNER') AND
BONUS = 500  ORDER BY WORKDEPT, SALARY
```

该查询从 EMPLOYEE 表返回 5 条记录，输出结果如下所示：

```
FIRSTNME      LASTNAME          WORKDEPT JOB       BONUS       SALARY
------------ --------------- -------- -------- ----------- -----------
MASATOSHI     YOSHIMURA        D11      DESIGNER  500.00      44680.00
MARILYN       SCOUTTEN         D11      DESIGNER  500.00      51340.00
BRUCE         ADAMSON          D11      DESIGNER  500.00      55280.00
KIYOSHI       YAMAMOTO         D11      DESIGNER  500.00      64680.00
IRVING        STERN            D11      MANAGER   500.00      72250.00
  5 record(s) selected.
```

使用下面的 RUNSTATS 命令更新统计信息：

```
RUNSTATS ON TABLE DB2INST1.EMPLOYEE
   WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

在没有列组统计信息的情况下再次收集统计信息，优化器会选择类似如下所示的查询访问计划，它的基数估计值为 2.428572。该查询访问计划如下所示：

```
$dynexpln -d sample -q "SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS,
SALARY  FROM EMPLOYEE  WHERE WORKDEPT IN ('D11','D21') AND   JOB IN
('MANAGER','DESIGNER') AND         BONUS = 500  ORDER BY WORKDEPT, SALARY" -t
Estimated Cost = 7.683424
Estimated Cardinality = 2.428572
Access Table Name = ORACLE.EMP1  ID = 3,12
|  #Columns = 5
|  Avoid Locking Committed Data
|  Evaluate Block/Data Predicates Before Locking Committed Row
|  Relation Scan
|  |  Prefetch: Eligible
|  Lock Intents..............................略..................................
```

在 WHERE 子句中应用了 3 个谓词，如果假设它们是独立的话，会导致优化器低估基数。为了解释说明优化器如何使用索引统计信息以及列组统计信息来检测统计关联，创建带有在谓词中引用的 3 个列(JOB、WORKDEPT、BONUS)的索引并收集统计信息。使用下面的命令创建索引并收集统计信息：

```
CREATE INDEX JOB DEPT BONUS ON EMPLOYEE (JOB,WORKDEPT,BONUS)
-- The RUNSTATS command provides the option to collect statistics on a set of
-- indexes only, without affecting the statistics previously collected.
RUNSTATS ON TABLE DB2INST1.EMPLOYEE FOR DETAILED INDEXES DB2INST1.JOB_DEPT_BONUS
```

然后优化器会使用新创建的索引以及在其上收集的统计信息来更正查询访问计划的基数估计值。从查询访问计划中更正了的基数估计值的输出如下所示：

```
Estimated Cost = 7.668263
Estimated Cardinality = 5.250000    -注：这个值和实际返回的 5 行相近
Table Constructor
|  2-Row(s)
Nested Loop Join
|  Access Table Name = ORACLE.EMP1  ID = 3,12
|  |  Index Scan:  Name = ORACLE.JOB_DEPT_BONUS  ID = 1
|  |  |  Regular Index (Not Clustered)
|  |  |  Index Columns:
|  |  |  |  1: JOB (Ascending)
|  |  |  |  2: WORKDEPT (Ascending)
|  |  |  |  3: BONUS (Ascending)
|  |  #Columns = 5.................................略.................................
```

C1=LITERAL1 OR C1=LITERAL2) AND (C2=LITERAL1 OR C2=LITERAL) AND C3=LITERAL

这个例子使用与前面等效的 OR 谓词来代替 IN 谓词。该 SQL 查询如下：

```
 SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS, SALARY       FROM EMPLOYEE
WHERE (WORKDEPT = 'D11' OR WORKDEPT = 'D21') AND           (JOB = 'MANAGER' OR
JOB = 'DESIGNER') AND           BONUS = 500     ORDER BY WORKDEPT, SALARY
```

该查询返回的结果与前面示例返回的结果相同。这个示例解释说明了部分统计信息对于优化器估计基数的能力的影响。执行下面的命令删除刚才创建的索引，并只使用组((JOB,WORKDEPT))上的列组统计信息再次收集统计信息：

```
DROP INDEX JOB DEPT BONUS
RUNSTATS ON TABLE DB2INST1.EMPLOYEE   ON ALL COLUMNS AND COLUMNS
     ((JOB,WORKDEPT))   WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

使用在合适的 IN、OR 和等式谓词所引用的列的子集上收集的列组统计信息，优化器估计出了更接近于真实结果的基数，但如果列组统计信息是在全部 3 个列上收集的话，该估计值的精确度要低于前一示例中的值。该查询访问计划输出如下所示：

```
Estimated Cost = 7.683424
Estimated Cardinality = 2.428572
Access Table Name = ORACLE.EMP1  ID = 3,12
|  #Columns = 5
|  Avoid Locking Committed Data
|  Evaluate Block/Data Predicates Before Locking Committed Row
|  Relation Scan
```

```
|  |  Prefetch: Eligible
|  Lock Intents
……………………略……………………
```

优化器使用了在组(JOB,WORKDEPT)上收集的列组统计信息来说明两个 OR 谓词间的统计关联，但不包括列组中的 BONUS，它认为 BONUS=500 谓词独立于那两个 OR 谓词，结果导致稍稍低估了最终的基数。

如果使用可视化解释或 db2exfmt 输出的“Optimized Statement”部分，会注意到该 SQL 语句的 OR 谓词被转换成了与它们等效的 IN 谓词。db2exfmt 的输出如下所示：

```
Optimized Statement:
-------------------
SELECT Q5.FIRSTNME AS "FIRSTNME", Q5.LASTNAME AS "LASTNAME", Q5.WORKDEPT AS
       "WORKDEPT", Q5.JOB AS "JOB", +0000500.00 AS "BONUS", Q5.SALARY AS
       "SALARY"FROM DB2INST1.EMPLOYEE AS Q5
WHERE (Q5.BONUS = +0000500.00) AND Q5.JOB IN ('MANAGER ', 'DESIGNER') AND
       Q5.WORKDEPT IN ('D11', 'D21') ORDER BY Q5.WORKDEPT, Q5.SALARY
```

在全部 3 列上收集列组统计信息会导致与前一示例相同的基数估计值。在这种情况下，仍然在前面的两列(JOB,WORKDEPT)上收集列组统计信息，并包括整个 3 列(JOB,WORKDEPT,BONUS)。RUNSTATS 命令如下所示：

```
RUNSTATS ON TABLE DB2INST1.EMPLOYEE   ON ALL COLUMNS AND COLUMNS
   ((JOB,WORKDEPT), (JOB,WORKDEPT,BONUS))   WITH DISTRIBUTION AND DETAILED
INDEXES ALL
```

可以在相同的列集合间收集到一个或更多的列组统计信息。收集这些统计信息后生成的查询访问计划与前一示例中最后的计划相同。至于这种情况的验证，就留给读者自己进行练习。

(C1=LITERAL1 AND C2=LITERAL2) OR (C1=LITERAL3 AND C2=LITERAL4)

这个例子解释说明了列组统计信息对合格的 OR 谓词的影响。考虑在 EMPLOYEE 表上执行如下查询。SQL 语句如下所示：

```
SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS, SALARY  FROM EMPLOYEE
WHERE ( WORKDEPT='E21' AND JOB='FIELDREP' ) OR  ( WORKDEPT='D21' AND
JOB='MANAGER' )  ORDER BY WORKDEPT, SALARY
```

该查询从 EMPLOYEE 表返回 6 条记录，输出结果如下所示：

```
FIRSTNME     LASTNAME          WORKDEPT JOB       BONUS          SALARY
```

```
------------ --------------- --------- -------- ----------- -----------
EVA          PULASKI          D21      MANAGER   700.00      96170.00
ROY          ALONZO           E21      FIELDREP  500.00      31840.00
HELENA       WONG             E21      FIELDREP  500.00      35370.00
RAMLAL       MEHTA            E21      FIELDREP  400.00      39950.00
JASON        GOUNOT           E21      FIELDREP  500.00      43840.00
WING         LEE              E21      FIELDREP  500.00      45370.00
 6 record(s) selected.
```

执行下面的 RUNSTATS 命令：

```
RUNSTATS ON TABLE DB2INST1.EMPLOYEE  WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

在没有列组统计信息的情况下再次收集统计信息，优化器会选择类似如下所示的查询访问计划，它的基数估计值为 1.880952。优化器选择的查询访问计划如下所示：

```
Estimated Cost = 7.718598
Estimated Cardinality = 1.880952
Access Table Name = ORACLE.EMP1  ID = 3,12
|  #Columns = 6
|  Avoid Locking Committed Data
|  |  Table: Intent Share..........................略....................................
```

在列(JOB,WORKDEPT)上收集列组统计信息能够让优化器更好地估计 OR 谓词的过滤效果，因为 OR 谓词的每一个子项都在 JOB 和 WORKDEPT 列上应用了一组本地等式谓词。执行下面的 RUNSTATS 命令：

```
RUNSTATS ON TABLE DB2INST1.EMPLOYEE   ON ALL COLUMNS AND COLUMNS
((JOB,WORKDEPT))  WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

收集了列组统计信息之后，优化器会选择类似如下所示的查询访问计划，它的基数估计值提高了，与实际的结果 6 行很接近。得到了更精确的基数估计值的查询访问计划，如下所示：

```
Estimated Cost = 128.489594
Estimated Cardinality = 5.600000
Access Table Name = ORACLE.EMPLOYEE  ID = 2,6
|  #Columns = 6
|  |  Prefetch: Eligible
..............................略.............................................
```

上面讲解了一些列组统计信息更新的示例，其实，优化器凭借精确的基数估计值来准

确计算出每个待定查询访问计划的成本。可以利用 DB2 中的列组统计信息的扩展用途来为优化器提供更多的信息，从而使优化器更好地估计基数，选择最佳的查询访问计划。列组统计信息对于处理复杂的 OLAP 查询尤其重要。

7.4 数据访问方式

查询优化器在编译一条 SQL 语句时估计满足查询的不同方法的执行成本。根据它的估计，优化器选择优化访问方案。访问方案指定解析一条 SQL 语句所需的操作顺序。当绑定一个应用程序时，会创建一个程序包。此程序包包含该应用程序中所有静态 SQL 语句的访问方案。动态 SQL 语句的访问方案在执行应用程序时创建。

有三种方法来访问表中的数据：

- 顺序扫描整个表。
- 通过首先访问表上的索引来定位特定表的行，再从表中读取相应行。
- 通过扫描共享。

可以根据谓词中定义的条件对行进行过滤(这些谓词通常在 WHERE 子句中指定)。所访问的表中被选中的行合并在一起生成结果集，还可以对输出的结果进行分组或排序等，作进一步处理。

从 DB2 9.7 开始，默认的数据访问方式是扫描共享。即，一个扫描可以使用另一个扫描的缓冲池页。扫描共享能够提高工作负载并发性和性能。通过进行扫描共享，系统能够支持更多的并发应用程序，查询可以更高效地执行，系统吞吐量得以提升，甚至连未参与扫描共享的查询也能从中受益。如果环境中的应用程序对大型表执行表扫描或多维集群(MDC)块索引扫描之类的扫描，那么扫描共享功能的效用尤其明显。编译器根据扫描类型、扫描目的、隔离级别以及对每条记录完成的工作量等因素来确定一个扫描是否适合参与扫描共享。

7.4.1 全表扫描

顺序扫描整个表叫作全表扫描(或者关系扫描)。顾名思义，全表扫描指的是在做查询的时候，为了得到满足谓词约束的结果需要遍历目标表的所有内容。显然这种数据访问方式在大多数情况下效率并不高，甚至很差。不难想象，如果目标表的数据内容很多(比如达到几百万或更多)，优化器经过评估之后最终选用了这种方式进行数据访问的话，带来的性能结果会是非常差的。但是对于一些记录很少的表，这种数据访问方式却可能会带来最好的性能表现，比如目标表的内容只有几十行的数据，那么与其先执行索引扫描再去表中访问数据以得到结果，还不如直接进行全表扫描的效率更高、性能更好。

所以在遇到全表扫描的时候并不能简单地得到绝对化的结论。这需要具体情况具体分析，下面的一些关于全表扫描的一些性能原则可以帮助我们更好地分析全表扫描的得失。

首先需要避免对较大的表进行全表扫描，但是在满足下面情况的时候可以考虑使用全表扫描：

- 从表中选择几乎所有的行
- 修改表中的所有行
- 表比较小

7.4.2 索引扫描

在数据库管理器因下列任何原因访问索引时，执行索引扫描：

- 在访问基本表前，缩小满足条件行的集合(通过扫描该索引特定范围内的行)。索引扫描范围(扫描的起点和终点)由比较索引列的查询中的值确定。
- 排序输出。
- 直接检索请求的列数据。如果所有请求的数据位于该索引中，那么不需要访问数据表。这称为完全索引访问(index only access)。

如果使用 ALLOW REVERSE SCANS 选项创建索引，那么也可以按与索引定义的方向相反的方向来扫描，也就是双向扫描。

如果未创建适当的索引或索引扫描的成本更高，那么优化器选择表扫描。当表不大，索引集群比率低或查询请求大多数表行时，索引扫描的成本可能更高。要查明访问方案使用表扫描还是索引扫描，使用解释工具(可参考本书“第 9 章：SQL 语句调优”的内容)。

定界范围的索引扫描

要确定索引是否可以用于特定查询，优化器从索引第 1 列开始对每列求值，以了解它是否可用来满足等式和 WHERE 子句中的其他谓词。谓词是 WHERE 子句中搜索条件的元素，表示或隐含比较运算。在下列情况下，可使用谓词来定界索引扫描范围：

- 测试是否等于常数、主变量以及值为常数的表达式或关键字。
- 测试“为空”还是“不为空”
- 测试是否等于基本子查询(是不包含 ANY、ALL 或 SOME 的子查询)，并且子查询没有相关列引用其直接父查询块(此子查询是其子选择的 SELECT)。
- 测试严格和相容不等式。

以下示例说明索引何时可以用来限制某个范围。

考虑具有下列定义的索引：

```
INDEX IX1:  NAME    ASC,
```

```
              DEPT     ASC,
              MGR      DESC,
              SALARY   DESC,
              YEARS    ASC
```

在此情况下，下列谓词可以用来限制索引 IX1 的扫描范围：

```
    WHERE  NAME  = :hv1
           AND  DEPT = :hv2
```

或者

```
    WHERE   MGR  = :hv1
            AND  NAME = :hv2
            AND  DEPT = :hv3
```

注意在第 2 个 WHERE 子句中，谓词不必按键列在索引中出现的相同顺序来指定。尽管示例使用绑定(宿主)变量，但其他变量(例如参数标记、表达式或常数)将会有相同的效果。

考虑使用 ALLOW REVERSE SCANS 参数创建的单个索引。这类索引支持以创建索引时定义的方向扫描和以相反的方向扫描。创建语句可能类似于如下所示：

```
    CREATE INDEX iname ON tname (cname DESC) ALLOW REVERSE SCANS
```

在这种情况下，根据 CNAME 中的降序值来构建索引(INAME)。允许反向扫描后，尽管列中的索引被定义为按降序扫描，但也可以按升序进行扫描。实际是否以两个方向使用索引不是由用户控制的，而是由优化器在创建和考虑访问方案时控制的。

在如下 WHERE 子句中，将只使用 NAME 和 DEPT 谓词来定界索引扫描的范围，而不使用 SALARY 或 YEARS 谓词：

```
    WHERE  NAME  = :hv1
            AND  DEPT = :hv2          AND  SALARY = :hv4
            AND  YEARS  = :hv5
```

这是因为有键列(MGR)将这些列与最前面两个索引键列(NAME，DEPT)分开，因此排序将关闭。但是，一旦该范围由 NAME = :hv1 和 DEPT = :hv2 谓词确定，那么可根据其余索引键列来对其余谓词求值。

测试不等式的索引扫描

某些不等式谓词可以对索引扫描的范围定界。有两种类型的不等式谓词：

(1) 严格不等式谓词

可用作范围定界谓词的严格不等式运算符是大于(>)和小于(<)。

只考虑一个具有严格不等式谓词的列用于定界索引扫描的范围。在下面这个示例中，NAME 和 DEPT 列上的谓词可用于定界该范围，但不能使用 MGR 列上的谓词。

```
WHERE  NAME  = :hv1
          AND  DEPT   > :hv2
          AND  DEPT   < :hv3
          AND  MGR    < :hv4
```

(2) 相容不等式谓词

以下是可用作范围定界谓词的相容不等式运算符：

- >= 和 <=
- BETWEEN
- LIKE

要定界索引扫描的范围，将考虑多个具有相容不等式谓词的列。在以下示例中，所有谓词都可用于定界索引扫描的范围：

```
WHERE  NAME  = :hv1
          AND  DEPT  >= :hv2
          AND  DEPT  <= :hv3
          AND  MGR   <= :hv4
```

要进一步说明此示例，假定“:hv2 = 404,:hv3 = 406 和:hv4 = 12345”。数据库管理器将使用该索引扫描部门 404 和 405 的全部，但当扫描到职员号(MGR 列)大于 12345 的第 1 个经理时，将停止扫描部门 406。

排序数据索引扫描

如果查询要求按排序顺序输出，并且如果排序列从第 1 个索引键列开始连续出现在该索引中，那么可使用索引来对数据排序。确定顺序或排序可由类似如下的操作产生：ORDER BY、DISTINCT、GROUP BY、“= ANY”子查询、“>ALL”子查询、“< ALL”子查询、INTERSECT、EXCEPT 或 UNION。一个例外是当比较索引键列是否等于“常数值”时，该常数值是值为常数的任何表达式。在这种情况下，排序列不能是第 1 个索引键列。

请考虑以下查询：

```
WHERE NAME = 'JONES'          AND DEPT = 'D93'      ORDER BY MGR
```

对于此查询，可使用索引来对行排序，因为 NAME 和 DEPT 将始终是相同的值，所以将被排序。也就是说，前导 WHERE 和 ORDER BY 子句等效于：

```
WHERE NAME = 'JONES'    AND DEPT = 'D93'  ORDER BY NAME, DEPT, MGR
```

也可使用唯一索引来降低排序顺序需求。考虑下列索引定义和 ORDER BY 子句：

```
UNIQUE INDEX IX0:  PROJNO  ASC
SELECT PROJNO, PROJNAME, DEPTNO         FROM PROJECT
     ORDER BY PROJNO, PROJNAME
```

不需要根据 PROJNAME 列再进行排序，因为 IX0 索引确保 PROJNO 是唯一的。此唯一性确保对于每个 PROJNO 值只有一个 PROJNAME 值。

本节说明了 SQL 编译器提供的两种数据访问方法：全表扫描和索引扫描。这两种方法在数据库进行实际的数据访问时通常都可能出现，在一般情况下进行较大规模数据访问的时候，索引扫描通常是首选方法，因为它能够提供更好的性能。在合理索引的配合下，索引扫描的性能一般都会高于甚至是远远高于全表扫描。

在使用索引扫描的时候，在目标表上创建合理的索引就显得十分重要，在访问同样数据的情况下，合理的索引与不合理的索引在性能上的差别可以达到几十倍、几百倍甚至更多。所以一定要创建合理的索引才能发挥出索引扫描的优势，如何构建出合理的索引请参考本书“第 8 章：索引设计和优化”。

7.4.3　扫描共享

扫描共享是指一个扫描利用另一扫描所完成工作的能力。共享的工作范围可以包括磁盘页读、磁盘查找、缓冲池内容复用以及解压等。

工作量繁重的扫描，例如对大型表进行的表扫描或多维集群(MDC)块索引扫描，有时适合与其他扫描共享页读。这样的共享扫描可以从表中的任意一点开始，以便利用已包含在缓冲池中的页。当共享扫描到达表的末尾时，它将从开头继续并在到达开始点时完成操作。

默认情况下，扫描共享功能处于启用状态，是否适合参与扫描共享由编译器自动确定。在运行时，符合条件的扫描可能参与，也可能不参与扫描共享，这取决于编译时某些内部的因素。

共享的扫描操作通过“共享组”进行管理。每个共享组尽可能将它们的成员置于一起，以便最大程度地增加共享的好处。如果共享组里一个扫描比另一个扫描快，那么页共享的好处可能会减弱或消失。在这种情况下，缓冲池中由第一个扫描访问的缓冲池页在共享组中的另一个扫描能够访问它们之前可能会被清除。数据服务器按同一个共享组中两个扫描之间的缓冲池页数测量它们之间的距离，数据服务器还监视扫描速度。如果同一个共享组中两个扫描之间的距离增大到太大，那么它们可能无法共享缓冲池页。为了减少这种情况，可以对较快的扫描进行调速，以允许较慢的扫描在数据页被清除前访问那些页。“共享集”通过同一种访问机制(例如表扫描或块索引扫描)访问同一个对象(例如一个表)的共享组的

集合。对于表扫描而言，页读顺序按页标识递增；对于块索引扫描而言，页读顺序按键值递增。

高优先级扫描操作从来不会被低优先级扫描降低扫描速度，而是移至另一个共享组。高优先级扫描可能会被放入某个组并由于该组中低优先级扫描者所完成的工作而受益。只要受益持续，它将一直停留在该组中。通过对快速扫描进行调速或者将其移至更快的共享组(如果该扫描遇到这样的组)，数据服务器不断调整共享组以确保共享保持最优。

可以使用 db2pd 命令的-scansharing 参数来查看关于扫描共享的信息。例如，对于单个共享扫描，db2pd 输出将显示扫描速度和扫描调节时间量之类的数据。对于共享组，db2pd 输出将显示组中的扫描数以及该组共享的页数。

7.5 连接方法

连接是根据信息的某些公共域从两个或多个表中组合信息的过程。当对应行中的信息符合连接条件时，一个表中的行就会与另一个表中的行配对。

在连接两个表时，无论使用哪种连接方法，总有一个表被选为外部表(OUTER TABLE)，而另一个表被选为内部表(INNER TABLE)。优化器根据所选连接方法的成本和类型决定哪个是外部表，哪个是内部表。首先访问外部表，并且只扫描一次。根据连接的类型和存在的索引，可以单次或多次扫描内部表。还有一点也很重要，即使试图连接两个以上的表，优化器每次也只连接两个表，并在必要时保存中间结果。

例如，考虑下面两个表：

Table1		Table2	
PROJ	PROJ_ID	PROJ_ID	名称
A	1	1	Sam
B	2	3	Joe
C	3	4	Mary
D	4	1	Sue
		2	Mike

要将标识列具有相同值的 TABLE1 和 TABLE2 连接，使用下列 SQL 语句：

```
SELECT PROJ, x.PROJ_ID, NAME    FROM TABLE1 x, TABLE2 y
  WHERE x.PROJ_ID = y.PROJ_ID
```

此查询得到下面一组结果行：

PROJ	PROJ_ID	名称
A	1	Sam
A	1	Sue
B	2	Mike
C	3	Joe
D	4	Mary

根据连接谓词是否存在，以及涉及的由表和索引统计信息来确定的各种成本，优化器选择下列一种连接方法：

- 嵌套循环连接
- 合并连接
- 哈希连接

可以提供显式连接运算符(如 INNER 或 LEFT OUTER JOIN)来确定如何在连接中使用表。但是，以这种方式改变查询之前，应允许优化器确定如何连接表，然后分析查询性能来决定是否要添加连接运算符。

7.5.1　嵌套循环连接

在嵌套循环连接中，外部表只被扫描一次。对于嵌套循环连接，要在内部表中找到与外部表中每一行相匹配的行，有两种方法：

(1) 扫描内部表。读取内部表中的每一行，并且针对该行决定是否应将其与正在考虑的外部表中的行相连接。

(2) 对内部表中的连接列进行索引查找。当用于连接的谓词所包含的列在内部表的索引中时，这种方法是可行的。这极大地减少了在内部表中访问的行数。

在嵌套循环连接中，决定哪个是外部表，哪个是内部表非常重要，因为外部表只扫描一次，而针对外部表中的每一行，都要访问一次内部表。正如前面提到的那样，优化器用成本模型来决定谁是外部表，谁是内部表。优化器做此决定时会考虑几个因素：

- 表的大小
- 缓冲池大小
- 谓词
- 排序要求
- 内部表上是否存在索引

嵌套循环连接是用下面两种方式中的一种执行的：

1) 对于外部表中每个被访问的行，扫描内部表。

例如，表 T1 和 T2 中的列 A 具有下列值：

外部表 T1：列 A	内部表 T2：列 A
2	3
3	2
3	2
	3
	1

要执行嵌套循环连接，数据库管理器执行下列步骤：

(1) 从 T1 中读取第 1 行。A 的值是“2”。

(2) 扫描 T2，直到发现一个匹配项(“2”)，然后连接这两行。

(3) 扫描 T2，直到发现下一个匹配项(“2”)，然后连接这两行。

(4) 扫描 T2，直至该表的结尾。

(5) 返回至 T1，并读取下一行(“3”)。

(6) 从第 1 行开始，扫描 T2，直到发现匹配项(“3”)，然后连接这两行。

(7) 扫描 T2，直到发现下一个匹配项(“3”)，然后连接这两行。

(8) 扫描 T2，直至该表的结尾。

(9) 返回至 T1，并读取下一行(“3”)。

(10) 像以前一样扫描 T2，连接匹配(“3”)的所有行。

2) 对于外部表中每个被访问的行，在内部表上执行索引查找。

如果存在下列形式的谓词，那么此方法可用于指定的谓词：

```
expr(outer_table.column)  relop  inner_table.column
```

其中，RELOP 是比较运算符(例如=、>、>=、< 或 <=)，而 EXPR 是基于外部表的有效表达式。请考虑以下示例：

```
OUTER.C1 + OUTER.C2 <= INNER.C1  OUTER.C4 < INNER.C3
```

此方法可以显著减少在每次访问外部表时要在内部表中访问的行数，虽然这取决于许多因素，包括连接谓词的选择性。

嵌套循环连接成本=访问第 1 个表的成本(COST) + 从第 1 个表中访问返回的行数×访

问第 2 个表的成本

从上面的公式中可以看到，如果访问第 1 个表时使用了全表扫描，那么嵌套循环连接的成本无疑是巨大的。

当对嵌套循环连接求值时，优化器在执行连接前也决定是否对外部表排序。如果根据连接列对外部表排序，那么可以减少从磁盘访问内部表的页的读操作次数，因为很可能这些页已在缓冲池中。如果连接使用高度集群的索引来访问内部表，并且如果外部表已排序，那么可将访问的索引页的数目减至最小。

7.5.2 合并连接

合并连接需要有等式连接谓词(具有 TABLE1.COLUMN=TABLE2.COLUMN 格式的谓词)。它还要求根据连接列对输入表进行排序。这称为等式连接谓词。合并连接需要连接列的已排序输入，这个输入可通过索引访问或通过排序来获得。通过扫描现有索引或在进行连接之前对表进行排序就可以做到这一点。连接列不能是 LONG 或 LOB 字段。

在合并连接中，同时扫描已连接的表。同时扫描两个表以查找匹配行。外部表和内部表都只扫描一次，除非外部表中有重复的值，那样的话可能要再次扫描内部表的某些部分。因为表通常只被扫描一次，所以决定哪个是外部表、哪个是内部表不像在其他连接方法中那么重要。尽管如此，由于可能有重复的值，因此，优化器通常选择重复值较少的表作为外部表。但是，优化器最终还是使用成本模型来决定谁是外部表，谁是内部表。

合并连接的外部表只扫描一次。除非外部表中出现重复的值，否则，内部表也只扫描一次。如果有重复的值出现，那么可能再次扫描内部表中的一组行。例如，如果表 T1 和 T2 中的列 A 有下列值：

外部表 T1：列 A	内部表 T2：列 A
2	1
3	2
3	2
	3
	3

要执行合并连接，数据库管理器执行下列步骤：

(1) 从 T1 中读取第 1 行。A 的值是“2”。

(2) 扫描 T2，直到发现匹配项，然后连接这两行。

(3) 连续扫描 T2，当列匹配时，连接那些行。

(4) 当读取 T2 中的“3”时，返回至 T1 并读取下一行。

(5) T1 中的下一个值是“3”，它与 T2 匹配，因此连接那些行。

(6) 连续扫描 T2，当列匹配时，连接那些行。

(7) 到达 T2 结尾。

(8) 返回至 T1 以读取下一行。注意 T1 中的下一个值与 T1 中的上一个值相同，因此从 T2 中的第一个“3”开始再次扫描 T2。数据库管理器记住此位置。

合并连接成本=访问第一个表的成本+访问第 2 个表的成本+排序的成本

7.5.3 哈希连接

哈希连接需要如下形式的一个或多个谓词：TABLE1.COLUMNX=TABLE2.COLUMNY，在该形式中，列类型相同。对于类型为 CHAR 的列，长度必须相同；对于类型为 DECIMAL 的列，精度和小数位必须相同；对于类型为 DECFLOAT 的列，精度必须相同。列类型不能是 LONG 字段列或大对象(LOB)列。哈希连接可处理多个等式谓词这一事实相对于合并连接是一大优势，后者只能处理等式谓词。

对于哈希连接，首先扫描内部表(也称为构建表，BUILD TABLE)，表中的行被复制到从排序堆划出的内存缓冲区中(该排序堆由 sortheap 数据库配置参数指定)。根据在 JOIN 谓词的列上计算的哈希值，将内存缓冲区分为若干分区。如果 INNER 表的大小超过可用的排序堆空间，那么将所选分区中的缓冲区写入临时表。然后扫描外部表(称为探测表，PROBE TABLE)。对于探测表中的每一行，对连接列应用同一哈希算法。如果所获得的哈希值与构建行的哈希值相匹配，就比较实际的连接列。如果与探测表行匹配的分区在内存中，那么比较会立即进行。如果分区被写入临时表，那么探测行也被写入临时表。最后，处理包含同一分区中的行的临时表以进行匹配。

由于将构建表保存在内存中所具有的好处，优化器通常选择较小的表作为构建表，以避免必须将该表溢出(SPILL)到磁盘上。但是，要再次强调的是，优化器成本模型最终决定哪个表是内部表、哪个表是外部表。

下面我们更深入地研究哈希连接如何利用 SMP 系统。在 SMP 系统(其中 INTRA_PARALLEL=ON 且 DFT_DEGREE>1)中，一个哈希连接可能由多个代理进程在同一个 CPU 或多个 CPU 上以并行方式执行。在以并行方式执行时，构建表被动态地分为多个并行的元组流，每个流由独立的任务处理以便将构建元组输入内存。在对构建表流的处理结束时，哈希连接进程会调整内存的内容，并执行任何必需的将分区移入或移出内存的操作。接下来，根据驻留内存的分区来处理探测表的多个并行元组流，并且在需要时，为溢出到临时表的哈希连接分区的元组来溢出这些并行元组流。最后，以并行方式处理溢出的分区，每个任务处理一个或多个溢出的分区。

为了获取哈希连接的所有性能效益，可能需要更改 sortheap 数据库配置参数和 SHEAPTHRES 数据库管理器配置参数的值。如果可以避免哈希循环和溢出到磁盘，哈希连接性能最佳。要调整哈希连接性能，估计对 sheapthres 可用的最大内存量，然后调整 sortheap 参数。增大它的设置，直到尽可能避免哈希循环和磁盘溢出，但不要达到由 sheapthres 参数指定的限制。增大 sortheap 值也应提高具有多个类别的查询的性能。

一般来说，哈希连接在 OLAP 中可以提高复杂 SQL 语句的性能，对于 OLTP 一般不建议使用哈希连接。

7.5.4　选择最佳连接的策略

到目前为止，我们已经讨论了在 DB2 中可用的不同连接方法。正如我们所知，初看起来，某些方法与其他方法相比是更好的选择。例如，与根据外部表的每一行扫描内部表的嵌套循环连接相比，合并连接具有只对表扫描一次的优势。于是，合并连接似乎是更好的选择；但是，如果存在索引的话，嵌套循环会是更好的选择。同样，与合并连接相比，哈希连接似乎是更好的选择，因为它不需要在执行前对输入表排序，但如果我们需要保持外部表中行的次序，合并连接或嵌套循环连接可能是更好的选择——哈希连接不能保证维持次序，因为它可能溢出到磁盘，而那样会破坏次序。

那么 DB2 优化器如何针对特定连接来决定使用哪种连接方法呢？首先，它必须考虑查询中谓词的类型。当选择了可能的连接方法时，DB2 优化器随后根据成本模型和选定的优化级别来决定使用哪种连接方法。优化级别是数据库配置文件中可配置的参数，它告诉优化器要进行多大程度的优化。这个值越高，优化操作就越多。优化级别可能的值为 0、1、2、3、5、7 和 9。这些值对可能的连接方法的影响如下：

- 嵌套循环连接在每个优化级别都可行。
- 合并连接在优化级别 1 及以上级别是可行的。
- 哈希连接在优化级别 5 及以上级别是可行的。

数据库优化器会根据成本模型来决定使用合适的连接。所以对我们来说只需要提供给优化器成本模型必需的信息：准确的统计信息、合理的配置参数、高效的 SQL 和索引以及优化级别。下面我们讲讲优化级别。

7.6　优化级别

优化级别指定了各种优化策略，当编译和优化 SQL 语句时，优化器将使用这些策略。连接方法的选择取决于正在使用的优化级别。所以，优化器并非总是使用上面描述的每种存取路径技术。相反，根据优化级别，优化器使用各种不同的技术。优化级别的用途是通

过它来指导 DB2 何时采用哪种优化策略和优化技术。通常，优化器考虑的优化策略越多，用于查询的存取方案就越好。然而，优化器被指导考虑的优化策略越多，把 SQL 编译成可执行的存取路径的准备时间就越长。幸运的是，可以设置优化级别来限制优化查询时应用的优化技术数目。对于较简单的查询、资源受限系统和动态 SQL，这是非常有用的。

7.6.1 优化级别概述

影响为 SQL 语句生成访问计划的方式的最重要因素就是优化级别，优化级别用于为此任务做准备。该信息告诉 DB2 优化器要付出多少努力、使用什么优化技术来确定解决查询的最佳访问计划。较高的级别将使优化器使用更为复杂的算法和代数分析——因而也需要花费更多的准备时间——来生成最终的访问计划。

DB2 优化级别的所有规则如图 7-4 所示。

	0	1	2	3	5	7	9
non-uniform distribution statistics: - most frequent values - quantiles	N	N	Y	most frequent values quantiles-No	Y		
Rewrite Rules	basic	subset	- all, except comp. intensive rules	- most sub-query-to-join transform.	- all, except computationally intensive rules		all
complex dynamic SQL queries					reduced depend on machine size and number of predicates	not reduced	
Search Strategies for Selecting Optimal Join	Greedy join 贪婪连接枚举，如果超过4个表，转为动态编程 if < 4 tables ==> greedy join = dynamic prog. join			动态编程连接枚举 :dynamic programming join limited use of - composite inner tables - Cartesian products for star schemas involving "lookup" tables			no limits
Join Concepts	NLJoin Index Scan	MSJoin NLJoin Index scan Table scan					
and MQT			HashJoin, MQT		HashJoin, MQT		
Star Join Strategy	N		Y				
List Prefetch	N		Y				
Index ORing	Y						
Index ANDing	N			Y			
Appropriate to use with:	OLTP		OLAP/DSS	for queries with 4 or more joins.			

图 7-4　DB2 优化级别的规则

连接枚举

连接枚举算法是对优化器使用的方案组合数的重要决定因子。优化器使用各种方法来选择查询的最佳连接策略。这些方法包括下列搜索策略，它们由查询的优化级别来确定：

(1) 贪婪连接枚举。

- 可节省空间和时间。
- 单向枚举，一旦为两个表选择了一个连接方法，在进一步优化期间就不更改它。
- 当连接多个表时，可能丢失最佳访问方案。如果查询只连接两三个表，那么贪婪连接枚举选择的访问方案与动态编程连接枚举选择的访问方案相同。如果该查询在相同的列上有多个连接谓词(显式指定的或通过谓词传递闭包隐式生成的)，那么更是如此。

(2) 动态编程连接枚举。

- 随着已连接表数的增加，空间和时间需求按指数规律增加。
- 可有效、全面地搜索以获得最佳访问方案。
- 类似于 DB2 z/OS 所使用的策略。

优化级别

当编译 SQL 或 XQuery 查询时，可以指定优化级别以确定优化器如何选择该查询的最有效访问方案。可使用查询编译中考虑的优化策略的编号和类型来区分优化级别。尽管可以单独指定优化技术以提高查询的运行时性能，但是指定的优化技术越多，查询编译就需要越多的时间和系统资源。

可以在编译 SQL 或 XQuery 查询时指定下列一个优化器级别：

级别 0：此级别指导优化器使用最少的优化来生成访问方案。此优化级别具有下列特征：

- 优化器不考虑任何非均匀分布统计信息。
- 仅应用基本的查询重写规则。
- 发生贪婪连接枚举。
- 仅允许使用嵌套循环连接及索引扫描访问方法。
- 在生成的访问方法中不使用列表预取。
- 不考虑星型连接策略。

此级别应该只用于需要最低的查询编译开销的情况。查询优化级别 0 适用于以下应用程序，完全由访问索引结构良好的表的非常简单的动态 SQL 或 XQuery 语句组成。

级别 1 - 此优化级别具有下列特征：

- 优化器不考虑任何非均匀分布统计信息。
- 只应用查询重写规则的一个子集。
- 发生贪婪连接枚举。
- 在生成的访问方法中不使用列表预取。

除了“合并扫描连接”及表扫描也可用以外，优化级别 1 类似于级别 0。

级别 2：此级别指导优化器使用比级别 1 显著高的优化程度，而使复杂查询的编译成

本显著低于级别 3 及更高级别。此优化级别具有下列特征：

- 利用了所有可用的统计信息，包括频率和分位数非均匀分布统计信息。
- 除只在极少情况下才适用的计算密集型规则外，将应用所有其他查询重写规则，包括路由对物化查询表的查询。
- 使用了贪婪连接枚举。
- 考虑各种访问方法，包括列表预取和物化查询表路由。
- 如果适用，请考虑星型连接策略。

优化级别 2 除了使用“贪婪连接枚举”而不是“动态编程”外，类似于级别 5。在所有使用“贪婪连接枚举”算法的级别中，此级别具有最高的优化程度。与级别 3 及更高级别相比，它对复杂查询的替代方案考虑较少，因而消耗的编译时间也少。建议将级别 2 用于决策支持或联机分析处理(OLAP)环境中非常复杂的查询。在这种环境下，特定查询很少完全重复，因此查询访问方案不太可能在高速缓存中停留到出现下一个查询为止。

级别 3：此级别请求中等优化。此级别与 DB2 MVS/ESA 版、OS/390 或 Z/OS 版的查询优化特征基本匹配。此优化级别具有下列特征：

- 使用非均匀分布统计信息(如果可用)，该统计信息跟踪频繁出现的值。
- 应用大部分查询重写规则，包括子查询至连接的变换。
- 动态编程连接枚举，如下所示：
  - 组合内部表的有限使用。
  - 涉及查找表的星型模式的笛卡尔乘积的有限使用。
- 考虑各种访问方法，包括列表预取、索引 AND 运算和星型连接。

此级别适用于大量应用程序。此级别改进具有 4 个或更多连接的查询的访问方案。但是，优化器可能无法考虑使用默认优化级别选择的更好方案。

级别 5：此级别指导优化器使用相当大量的优化来生成访问方案。此优化级别具有下列特征：

- 使用所有可用的统计信息，包括频率和分位数分布统计信息。
- 除只在极少情况下才适用的那些计算密集型规则外，将应用所有其他查询重写规则，包括路由对物化查询表的查询。
- 动态编程连接枚举，如下所示：
  - 组合内部表的有限使用。
  - 涉及查找表的星型模式的笛卡尔乘积的有限使用。
- 考虑各种访问方法，包括列表预取、索引 AND 运算和物化查询表路由。

当优化器检测到不能保证用于复杂动态 SQL 或 XQuery 查询的其他资源和处理时间时，将减少优化。减少的范围或大小取决于机器大小和谓词数目。

当查询优化器减少查询优化量时，它继续应用正常时应用的所有查询重写规则。但是，它的确使用了贪婪联合枚举法并减少了考虑的访问方案的组合数。

对于由事务和复杂查询组成的混合环境，查询优化级别 5 是很好的选择。此优化级别设计成可以用高效的方式应用最有价值的查询变换和其他查询优化技术。

级别 7：此级别指导优化器使用相当大量的优化来生成访问方案。级别 7 除了不减少用于复杂动态 SQL 或 XQuery 查询的查询优化量外，它与查询优化级别 5 是相同的。

级别 9：此级别指导优化器使用所有可用的优化技术。这些主要功能包括：

- 所有可用的统计信息。
- 所有查询重写规则。
- 联合枚举的所有可能性，包括笛卡尔乘积和任意多种组合的内部结构。
- 所有访问方法。

此级别可以大大扩展由优化器考虑的可能访问方案的数量。对于使用大表的很复杂且运行时间很长的查询，可以使用此级别来确定更全面优化是否将生成更好的访问方案。使用解释工具和基准测试工具来验证是否实际上已找到更好的方案。

7.6.2　选择优化级别

合理地设置优化级别可以提供有关显式指定优化技术的某些优点，特别是由于下列原因：

- 管理非常小型的数据库或非常简单的动态查询。
- 适应数据库服务器上编译时的内存限制。
- 减少查询编译时间，如 PREPARE 时间。

通过使用优化级别 5(默认查询优化级别)，大多数语句都可以使用合理数量的资源充分优化。以给定的优化级别，查询编译时间和资源消耗主要受查询的复杂性，特别是连接和子查询的数量影响。但是，编译时间和资源的使用也受所执行的优化数量影响。

查询优化级别 1、2、3、5 和 7 都比较通用。仅当需要进一步减少查询编译时间且了解 SQL 和 XQuery 语句相当简单时，才考虑级别 0。

选择优化级别时的原则：

当选择优化级别时，考虑下列一般准则：

- 通过使用默认查询优化级别(级别 5)开始。
- 要使用不是默认值的级别，首先尝试级别 1、2 或 3。级别 0、1 和 2 使用贪婪连接枚举算法。
- 如果有许多表，这些表的同一列有许多连接谓词，并且又关心编译时间，那么使用优化级别 1 或 2。

- 对于运行时间短(不到 1s)的查询，使用低优化级别(0 或 1)。这些查询趋向于具有下列特征：
  - 访问单个表或仅访问少量表。
  - 访存一行或仅访存少量行。
  - 使用标准的唯一索引。

这种查询的典型示例是联机事务处理(OLTP)事务。

- 对于运行时间较长(大于 30 秒)的查询，使用高优化级别(3、5 或 7)。
  - 级别 3 和更高的级别使用“动态编程”连接枚举算法。与级别 0、1 和 2 相比，此算法考虑更多替代方案，并且可能使编译时间也显著增加，特别是随着表的数量增加，更是如此。
  - 仅当具有查询的特定异常的优化需求时，使用优化级别 9。

复杂的查询可能需要不同程度的优化，以便选择最佳访问方案。对具有下列特征的查询考虑使用更高的优化级别：

- 访问大型表
- 大量谓词
- 很多子查询
- 很多连接
- 很多集合运算符，例如 UNION 和 INTERSECT
- 很多限定行
- GROUP BY 和 HAVING 操作
- 嵌套表表达式
- 大量视图

复杂查询的典型例子是对完全规范化数据库的决策支持查询或月结报告查询，对于这些复杂查询，至少应该使用默认查询优化级别。要分析长时间运行的查询，可使用 db2batch 运行该查询以确定编译和执行各花费了多长时间。如果编译需要更长时间，那么降低优化级别。如果执行需要更长时间，那么考虑更高的优化级别。

对于由查询生成器产生的 SQL 和 XQuery 使用较高的查询优化级别。许多查询生成器创建运行效率不高的查询。编写很差的查询，包括那些由查询生成器产生的查询，需要另外优化以选择良好的访问方案。使用查询优化级别 2 和更高的级别可以改进这样的 SQL 和 XQuery 查询。

7.6.3 设置优化级别

当指定优化级别时，考虑查询是使用动态 SQL 和 XQuery 语句还是静态 SQL 和 XQuery

语句，以及是否重复执行同一动态查询。对于静态 SQL 和 XQuery 语句，只耗费一次查询编译时间和资源，而产生的方案可以使用许多次。一般来说，静态 SQL 和 XQuery 语句应该始终使用默认查询优化级别。因为动态语句是在运行时绑定和执行的，所以考虑对动态语句进行另外的优化开销是否改善总体性能。但是，如果重复执行相同的动态 SQL 或 XQuery 语句，那么高速缓存所选访问方案。这种语句可以使用与静态 SQL 或 XQuery 语句相同的优化级别。

如果认为查询可以从附加优化中获益，但不能肯定，或者关心编译时间和资源使用情况，那么可以执行一些基准程序测试。

要设置查询优化级别，遵循下列步骤：

(1) 按以下方式非正式地或正式地测试分析性能因素：

- 对于动态查询语句，测试应比较该语句的平均运行时间。使用下列公式来估计平均运行时间：

```
         编译时间 + 所有重复的执行时间的总和
        ------------------------------------------------------------
                     重复次数
```

在此公式中，重复次数表示每次编译查询语句时，期望查询语句执行的次数。

注意：

在初始编译之后，当环境更改要求动态 SQL 和 XQuery 语句时，将重新编译这些动态 SQL 和 XQuery 语句。如果在将查询语句高速缓存之后环境不更改，那么不需要再次编译，因为后来的 PREPARE 语句将复用高速缓存的语句。

- 对于静态 SQL 和 XQuery 语句，比较语句运行时间。

尽管可能对静态 SQL 和 XQuery 语句的编译时间也感兴趣，但该语句的总计编译与运行时间在任何有意义的上下文中都很难估计。比较总时间不能识别这样一个事实，即每次将静态查询语句绑定时，它可以运行多次，而在运行时，一般不将它绑定。

(2) 可以在数据库、应用程序和会话级别设置优化级别：

- 动态 SQL 和 XQuery 语句使用 CURRENT QUERY OPTIMIZATION 专用寄存器指定的优化级别，该寄存器是用 SQL 语句 SET 设置的。例如，下列语句将优化级别设置为 1：

```
DB2 SET CURRENT QUERY OPTIMIZATION = 1
```

要确保动态 SQL 或 XQuery 语句始终使用同一优化级别，可以将 SET 语句包含在应用程序中。

如果尚未设置 CURRENT QUERY OPTIMIZATION 寄存器，将使用默认查询优化级别绑定动态语句。动态和静态查询的默认值由数据库配置参数 dft_queryopt 的值确定。级别 5 是此参数的默认值。绑定选项和专用寄存器的默认值也是从 dft_queryopt 数据库配置参数读取的。

- 静态 SQL 和 XQuery 语句在应用程序中使用 PREP 和 BIND 命令上指定的优化级别。SYSCAT.PACKAGES 目录表中的 QUERYOPT 列记录用于绑定程序包的优化级别。如果以隐式方式或使用 REBIND PACKAGE 命令重新绑定程序包，将把同一优化级别用于静态查询语句。要更改这种静态 SQL 和 XQuery 语句的优化级别，可使用 BIND 命令。如果未指定优化级别，那么 DB2 使用 dft_queryopt 数据库配置参数指定的默认优化级别。
- 如果没有在会话和应用程序级别制定优化级别，那么默认将使用数据库级别的优化级别，可以设置数据库配置参数 dft_queryopt 来更改数据库的优化级别，这个参数是全局级的，会影响数据库中的所有应用程序和 SQL。下面是设置数据库优化级别为 7 的示例：

```
db2 update db cfg for sample using dft_queryopt 7
```

7.7 基于规则的优化

7.7.1 优化器概要文件概述

大多数主流关系数据库管理系统，例如 DB2、Oracle、Informix 和 SQL Server 都依赖于基于成本的优化器设计，在数据库服务器环境中的一组经常变化的条件(包括变化的查询特征和数据)的影响下，从很多可能的计划中选择最佳 SQL 执行计划。具体而言，DB2 数据库 SQL 优化的决定受系统配置(I/O 存储特征、CPU 并行性和速度、缓冲池和排序堆设置、通信带宽)、模式(索引、约束)、DB2 注册变量、DB2 优化级别和统计信息(关于表、列和索引的统计信息)的影响。这么多复杂的因素，再加上数据本身的动态性，使得最佳计划的评估对于任何数据库系统而言通常都是复杂的过程。

很多人都曾在应用程序中碰到这样的情况：大多数查询工作负载都经过了适当的调优，并取得了较好的性能，但是随着用户期望的增长，加上系统的复杂性和多样性，仍然有少许 SQL 语句无法通过调优取得预期的性能。虽然人们已经尽了最大的努力，试图通过改变数据库(例如使用索引建议器或其他方法来改进索引、更新统计信息、改善数据群集及更改参数)来调优 SQL 语句，但是问题仍然存在。有时候，我们希望更直接地影响优化器，同时尽量避免更改应用程序。

这时候我们可以考虑使用基于规则的优化器——优化概要文件。然而需要注意的是，先进的优化器在生成特定的访问计划时，必然有其原因，所以在应用优化概要文件之前，务必理解是什么原因导致查询的性能低下。优化概要文件使用起来并不难，但更具有挑战性的任务是根据给定的数据库环境判断 SQL 语句的问题出在哪里，并选择适当的优化概要加以应用。

DB2 优化概要文件可以将规则传递给优化器，用于指导优化器为 SQL 查询生成所需的执行计划，以覆盖默认的成本模型。这一点类似于 Oracle 和 Informix 数据库中的 RBO 优化器(在 SQL 语句中应用 HINT)。优化概要文件是包含一个或多个 SQL 语句的优化准则的 XML 文档。通过使用 SQL 文本和明确标识 SQL 语句所需的其他相关信息建立每个 SQL 语句与其关联的优化准则之间的通信。

优化概要文件的工作原理

首先选择一组想要影响其访问计划的查询。然后，将这些查询和一些适当的指南放到 XML 优化概要文件中。为了通过验证，XML 优化概要文件必须遵从优化指南 XML 模式，并由一些区段组成。下面的 XML 优化概要文件演示了如何使用优化概要文件将优化准则传递给 DB2 优化器：

```
<?xml version="1.0" encoding="UTF-8"?>
<OPTPROFILE VERSION="10.5.0.6">
<!-- Optional zero or one Global section -->
<OPTGUIDELINES>
   <MQT NAME="DB2ADMIN.MQT1"/>
</OPTGUIDELINES>
<!-- Zero or more Statement profile section.-->
  <STMTPROFILE ID="Query 0">
     <STMTKEY SCHEMA="DB2ADMIN">
        <![CDATA[select c2 from t1 where c1=100]]> -注：要优化的 SQL 语句
     </STMTKEY>
     <!-- Guideline for Query 0-->
     <OPTGUIDELINES>
        <IXSCAN TABID="T1" INDEX="T1X"/>  -注：优化规则，对表 Q1 使用索引 T1X 扫描
     </OPTGUIDELINES>
  </STMTPROFILE>
</OPTPROFILE>
```

XML 优化概要文件以 OPTPROFILE 区段开始，该区段表明版本属性。这个全局区段将规则全局地应用到所有 SQL 语句上。例如，可以指定使用哪个 REOPT 选项，使用哪个 MQT 表，或者使用什么样的查询优化。每个 STMTPROFILE 元素都为应用程序语句提供

一组优化准则。STMTPROFILE 区段则表明将哪些特定的规则应用于 STMTKEY 元素中的 SQL 语句上。

如果有问题的 SQL 查询不容易访问到，那么借助 XML 优化概要文件可以带来很大的方便。例如，SQL 查询可能处在一个应用程序中，而这个应用程序是不能更改的。在这种情况下，可以使用概要文件，在查询文本成功匹配之后，通过触发与查询相关联的优化概要文件来影响查询行为。该环境中的所有 SQL 语句将尝试从活动的优化概要文件中查找匹配项，而这种匹配是高效率、低开销的。

7.7.2 启用优化概要文件

一个数据库中可以有很多个优化概要文件，但是在实际情况中，更灵活的做法是创建一个主优化概要文件，将所有规则(STATEMENT PROFILE)组织在一起，然后只激活此概要文件，根据应用程序环境的不同，可以选择以下几种方法之一来激活优化概要文件。另外，还需要将 DB2_OPTPROFILE 注册变量设置为 YES。

1) 在 CLP 环境中

使用"SET CURRENT OPTIMIZATION PROFILE=DB2ADMIN.PROF1"语句在会话级将概要文件与所有 SQL 语句关联，直到连接重置或概要文件重置。这条语句还可以嵌入到应用程序中。

2) 对于 CLI 应用程序或使用旧的 JDBC 驱动程序的 JDBC 应用程序

在 DB2CLI.INI 配置文件中设置 CURRENTOPTIMIZATIONPROFILE 关键字来关联优化概要文件。例如对于 SAMPLE 数据库，这个关键字是在 DATA SOURCE 区段中设置的，如下所示：

```
[SAMPLE]
CURRENTOPTIMIZATIONPROFILE=DB2ADMIN.PROF1
```

经过这样设置后，应用程序执行中的 SQL 将尝试与 DB2ADMIN.PROF1 中的 SQL 语句进行匹配，查找指定的规则，这些规则将覆盖执行环境中常规的优化。

3) 对于使用 JCC UNIVERSAL DRIVER 的 JDBC 应用程序

采用 JCC UNIVERSAL DRIVER 的 JDBC 应用程序并不使用 DB2 CLI 层。虽然可以将系统包和绑定文件与动态 SQL 执行相关联，但最好的做法是将"SET CURRENT OPTIMIZATION PROFILE"语句嵌入在 Java 应用程序中，在会话级关联概要文件。

4) 对于 SQL PL 过程

在创建 SQL PL 过程之前，使用 SET_ROUTINE_OPTS 过程调用将优化概要文件的名

称与 DB2 中特定的 SQL PL 相关联。如下所示：

```
CALL SYSPROC.SET_ROUTINE_OPTS('OPTPROFILE  DB2ADMIN.PROF1')
```

SQL PL 过程包含的 SQL 语句具有一些执行属性，例如隔离级别或优化级别，这些属性只能通过 DB2_SQLROUTINE_PREOPTS 注册变量来覆盖。也可以用 SYSPROC.SET_ROUTINE_OPTS 过程覆盖该选项。要激活概要文件，可以使用该存储过程来关联优化概要文件。

5) 对于 C/C++应用程序中的嵌入式 SQL

对于嵌入式 C/C++应用程序，使用 OPTPROFILE 绑定选项。嵌入式 SQC 程序需要使用 PREP 命令来编译，该命令将创建绑定文件。这个绑定文件需要通过 OPTPROFILE 选项绑定到数据库，例如执行下面的命令：

```
db2 bind prog1.bnd OPTPROFILE DB2ADMIN.PROF1
```

6) 对于含嵌入式静态 SQL 语句的 SQLJ Java 应用程序

在定制阶段使用 BINDOPTIONS 参数关联概要文件。静态 SQLJ 程序 PROG1 按如下所示进行翻译和编译：

```
sqlj prog1.sqlj
db2sqljcustomize -url jdbc:db2://SERVER:PORT/SAMPLE -user USER -password
PASSWORD -bindoptions "OPTPROFILE DB2ADMIN.PROF1" -storebindoptions
prog1_SJProfile0
```

所有使用旧的 JDBC 驱动程序的 JDBC 程序，都将使用 DB2CLI.INI 中的设置。使用 UNIVERSAL JDBC 驱动程序的 JDBC 程序属于上述第(3)类情况。需要注意的是，由于 SQLJ 为 SELECT SQL 语句生成隐式的“DECLARE CURSOR”子句，因此为了使优化概要文件得到应用，优化概要文件除了包括 SELECT 语句外，还需要包括“DECLARE CURSOR”子句。

当应用程序执行时，将 SQL 与活动的概要文件中的规则相比较。如果存在匹配的 STMTKEY，优化规则就会开始起作用；反之，假如优化规则被认为是不适用的或无效的，那么会返回 RC=13 的 SQL0437W。DB2 解释工具对于帮助确定优化规则是否被选择非常有用。解释工具的输出会指明优化概要文件的名称和有效的优化规则。概要文件中的优化规则通常覆盖用于应用程序设置的常规优化，从而使概要文件能够更好地控制计划评估。

7.7.3　优化概要文件使用示例

优化概要文件中的任何优化规则都必须遵从 DB2 提供的 XML 模式。如果没有正确地

指定优化规则，那么优化规则将无效，并且在大多数情况下，将返回 RC=13 的 SQL0437W。启用优化概要文件前必须设置 db2_optprofile 注册变量。此变量是 DB2 内部变量，不能通过“db2set –lr”命令看到，不过在设置成 YES 之后可以通过“db2set –all”命令看到。优化概要文件存储在名为 systools.opt_profile 的表中。如果从这个表中更新或删除指南，那么需要通过发出 FLUSH OPTIMIZATION PROFILE CACHE 语句更新缓存，使之可以被使用。需要注意的是，SQL 语句测试匹配是大小写敏感的，但在尝试匹配之前，DB2 将去除冗余空格和控制字符。

可以通过下面两种方法来创建该表：

(1) 调用 sysinstallobjects 过程：

```
db2 "call sysinstallobjects('opt_profiles', 'c', '', '')"
```

(2) 发出 CREATE TABLE 命令：

```
CREATE TABLE SYSTOOLS.OPT PROFILE (
SCHEMA  VARCHAR(128) NOT NULL, -注：优化概要文件的模式限定符。
NAME    VARCHAR(128) NOT NULL, -注：指定优化概要文件名称，不能超过 128 个字符。
PROFILE BLOB (2M)    NOT NULL,-注：用于定义优化概要文件的 XML 文档
PRIMARY KEY ( SCHEMA, NAME ));-注：表的主键
```

假设我们准备使用的优化概要为 DB2ADMIN.PROF1，存放 XML 的文件为 DB2ADMIN.PROF1.XML，创建的 PROFILEDATA 文件将包含以下内容：

```
"DB2ADMIN","PROF1","DB2ADMIN.PROF1.xml"
```

然后使用下面的 IMPORT 命令将 PROFILEDATA 文件中的内容导入到表 SYSTOOLS.OPT_PROFILE 中：

```
IMPORT FROM profiledata OF DEL MODIFIED BY LOBSINFILE INSERT UPDATE INTO
SYSTOOLS.OPT_PROFILE
```

如果希望删除不需要的优化概要文件，使用正常的 DELETE 语句删除即可。

由于下面的示例都是使用 DB2 命令行工具进行的，因此需要进行以下设置，让 DB2 CLP 使用优化概要文件 DB2ADMIN.PROF1：

```
$db2 SET CURRENT OPTIMIZATION PROFILE DB2ADMIN.PROF1
```

下面的例子演示了优化概要文件在(3)类情况下的使用——常规优化、查询重写和计划优化。

例 7-1　总是使用索引 T1X(计划优化)。

假设在表 T1 的(C2, C1)列上有一个索引 T1X。根据优化器的成本计算，对于以下查询，会导致表扫描。下面的代码展示了如何强制使用索引：

```
<STMTPROFILE ID="Guideline for query 1">
  <STMTKEY SCHEMA="KCHEN">
    <![CDATA[select c2 from t1 where c1=?]]>
  </STMTKEY>
  <OPTGUIDELINES>
    <IXSCAN TABID="T1" INDEX="T1X"/>
  </OPTGUIDELINES>
</STMTPROFILE>
```

例 7-2　总是使用 REOPT(常规优化)。

可以使用 REOPT 优化规则，将查询优化推迟到运行时输入变量已知的时候。可能的选项有 ONCE、ALWAYS 或 NONE，如下所示：

```
<STMTPROFILE ID="Guideline for query 2">
   <STMTKEY>
     <![CDATA[select c4 from t1 where c1 = ?]]>
   </STMTKEY>
   <OPTGUIDELINES>
    <REOPT VALUE='ALWAYS'/>
  </OPTGUIDELINES>
</STMTPROFILE>
```

例 7-3　只使用 DB2 中的“Optimization Level 0”(常规优化)。

通常，对于应用程序而言，优化级别是固定的，但是如果要使一条特定的 SQL 语句在不同的优化级别上执行，那么可以创建以下优化概要文件：

```
<STMTPROFILE ID="Guideline for query 3">
   <STMTKEY>
     <![CDATA[select * from t1 where c1 = 2]]>
   </STMTKEY>
   <OPTGUIDELINES>
     <QRYOPT VALUE="0"></QRYOPT>
   </OPTGUIDELINES>
</STMTPROFILE>
```

例 7-4　只使用 DB2 中的“RUNTIME DEGREE ANY”(常规优化)。

可以有很多方法来修改内部分区的查询的运行时等级。下面的代码展示了优化概要文

件如何为查询指定运行时等级以及如何影响查询的执行：

```
<STMTPROFILE ID="Guideline for query 4">
   <STMTKEY>
     <![CDATA[select * from t1 where c1 = 3]]>
   </STMTKEY>
   <OPTGUIDELINES>
     <DEGREE VALUE="ANY"></DEGREE>
   </OPTGUIDELINES>
</STMTPROFILE>
```

例 7-5　INLIST 改为嵌套循环连接(查询重写)。

将值列表(INLIST)改为使用 GENROW 函数非常有效，可以提高查询的性能。在这个例子中，值列表放在内存的一个表中：

```
 <STMTPROFILE ID="Guideline for query 5">
   <STMTKEY>
     <![CDATA[SELECT S.S NAME, S.S SUPPKEY, PS.PS PARTKEY,
                 P.P SIZE, P.P TYPE, S.S NATION
            FROM KCHEN.PARTS P, KCHEN.SUPPLIERS S, KCHEN.PARTSUPP PS
            WHERE P PARTKEY = PS.PS PARTKEY AND
                 S.S SUPPKEY = PS.PS SUPPKEY AND
                 P.P TYPE IN ('BRASS', 'BRONZE') AND
                 P.P SIZE IN (31, 31, 33, 34) AND
                 S.S NATION = 'PERU']]>
   </STMTKEY>
   <OPTGUIDELINES>
     <INLIST2JOIN TABLE='P' COLUMN="P TYPE" OPTION='ENABLE'/>
   </OPTGUIDELINES>
</STMTPROFILE>
```

例 7-6　子查询改为连接(查询重写)。

在这个例子中，在查询重写期间，通过使用带 ENABLE 属性的 SUBQ2JOIN，将子查询转换成连接，以便更好地对其进行优化：

```
 <STMTPROFILE ID=" Guideline for query 6">
   <STMTKEY>
      <![CDATA[SELECT PS.PS PARTKEY,  COUNT(DISTINCT PS.PS SUPPKEY)
            FROM KCHEN.PARTSUPP PS, KCHEN.LINEITEM
            WHERE PS.PS PARTKEY = L PARTKEY AND
                 PS.PS_PARTKEY = ANY (
```

```
                                                    SELECT P PARTKEY FROM KCHEN.PARTS
                                                    WHERE P BRAND <> 'Brand#45' AND
                                                    P NAME='peach snow puff bisque misty'
                                                    AND P_TYPE <> 'TIN')
                   GROUP BY PS PARTKEY]]>
         </STMTKEY>
         <OPTGUIDELINES>
            <SUBQ2JOIN OPTION="ENABLE" />
         </OPTGUIDELINES>
      </STMTPROFILE>
```

例 7-7　影响连接顺序(计划优化)。

通常，查询的连接顺序在很大程度上决定了查询的执行性能，因为越早地过滤行，效率越高。可以使用以下优化规则来影响连接顺序。注意，当出现多个表引用时，使用 TABLEID 属性而不是 TABID 属性：

```
<STMTPROFILE ID="Guideline for query 7">
  <STMTKEY>
    <![CDATA[select * from T4 t74, T3 t73, T2 t72, T1 t71
            where t71.c1 = t72.c1 and
                  t72.c2 = t74.c2 and
                  t74.c1 = t73.c1 and
                  t73.c2 = t71.c2 and
                  t71.c3 = t74.c3 and
                  t72.c3 = t73.c3]]>
  </STMTKEY>
  <OPTGUIDELINES>
    <MSJOIN FIRST="TRUE" outermost="true">
       <ACCESS TABLEID="t71"/>
<MSJOIN>
    <ACCESS TABLEID="t73"/>
    <ACCESS TABLEID="t74"/>
       </MSJOIN>
    </MSJOIN>
   </OPTGUIDELINES>
</STMTPROFILE>
```

例 7-8　强制使用索引扫描。

对下面的语句由于是从 EMP1 中选择全部数据，因而正常情况下会进行全表扫描：

```
SELECT * FROM EMP1
```

在仿造 EMPLOYEE 表对 EMP1 在 EMPNO 列上建立主键索引后，就可以使用下面的优化概要文件强制该 SQL 先走 EMP1 的主键索引，然后再根据 RID 读取数据：

```
<?xml version="1.0" encoding="UTF-8"?>
<OPTPROFILE version="10.5.0.6">
<STMTPROFILE id="testoptprofile">
<STMTKEY SCHEMA="DB2ADMIN">
   SELECT * FROM EMP1   -注：SQL 语句必须匹配，注意大小写和空格
</STMTKEY>
<OPTGUIDELINES>
 <IXSCAN TABLE="EMP1" />   -注：要求使用索引进行扫描
</OPTGUIDELINES>
</STMTPROFILE>
</OPTPROFILE>
```

使用 db2exfmt 得到以下执行计划:

```
Profile Information:
--------------------
OPT PROF: (Optimization Profile Name)
 DB2ADMIN.PROF1                -注：使用的概要文件
……………省略……………………

        /----+---\
      42          42
    IXSCAN    TABLE: DB2ADMIN     -注：使用索引扫描
    (   3)        EMP1
   0.0375529
      0
      |
      42
 INDEX: DB2ADMIN
 CC1230805503781
```

例 7-9　强制使用表扫描。

当执行下面的语句时，由于 EMP1 的 EMPNO 列存在主键索引，因此正常情况下会进行索引扫描：

```
SELECT * FROM EMP1 WHERE EMPNO='000010'
```

使用下面的优化概要文件，要求优化器直接使用表扫描：

```
<?xml version="1.0" encoding="UTF-8"?>
<OPTPROFILE version="10.5.0.6">
<STMTPROFILE id="testoptprofile">
<STMTKEY SCHEMA="DB2ADMIN">
   SELECT * FROM EMP1 WHERE EMPNO='000010'
</STMTKEY>
<OPTGUIDELINES>
 <TBSCAN TABLE="EMP1" /> -注：要求使用表扫描
</OPTGUIDELINES>
</STMTPROFILE>
</OPTPROFILE>
```

使用 db2exfmt 得到以下执行计划：

```
Profile Information:
--------------------
OPT PROF: (Optimization Profile Name)
 DB2ADMIN.PROF1            -注：使用的概要文件
..................省略........................
      Rows
     RETURN
     (   1)
      Cost
       I/O
       |
        1
     TBSCAN        -注：使用表扫描
     (   2)
     7.60795
        1
       |
       42
 TABLE: DB2ADMIN
      EMP1
```

上面举了一些使用优化概要文件的例子，其实 DB2 的优化器无比强大，通常情况下很少有机会用到优化概要文件，而且要强调的是优化概要文件不是万能药，只是止痛药。只有在万不得已的情况下才使用优化概要文件。一般来说，对于优化器问题，用户应该尽量找到根本原因(root cause)，而不是简单地指定优化概要文件了事。

7.8 如何影响优化器来提高性能

我们已经理解了优化器的工作原理和组件，优化器能否正常高效工作取决于我们需要提供的影响优化器工作的如下重要性能准则。因此，只要记住这些影响 DB2 优化器的准则，就可以实现这些准则以获得更好的 SQL 性能。

7.8.1 使 DB2 统计信息保持最新

如果没有存储在 DB2 系统目录中的统计信息，优化器在优化任何事物时都会遇到困难。这些统计信息向优化器提供了与正在被优化的 SQL 语句将要访问的表和索引相关的信息。同时当数据分布不均匀时考虑收集分布统计信息，当存在多个谓词相关时考虑收集列组统计信息。

7.8.2 构建适当的索引

也许为保证最佳 DB2 应用程序性能可以做的最重要的事就是根据应用程序使用的查询创建正确的索引。创建的索引不是多多益善，而是要遵循一些创建索引的原则。例如，考虑以下这条 SQL 语句：

```
SELECT   LASTNAME, SALARY
FROM     EMP
WHERE    EMPNO = '000010'
AND      DEPTNO =  'D01'
```

什么索引会对这个简单查询有作用？首先，考虑可以创建的所有可能的索引。第 1 个简短列表可能看起来如下：

- EMPNO 上的 INDEX1
- DEPTNO 上的 INDEX2
- EMPNO 和 DEPTNO 上的 INDEX3

这是一个好的开始，INDEX3 可能是最好的。它让 DB2 使用索引来立即查找满足 WHERE 子句中的两个简单谓词的行。当然，如果已经有许多关于 EMP 表的索引，也许应该检查再创建另一个关于表的索引所带来的影响。要考虑的因素包括：

- **修改影响**：DB2 将自动维护每个索引。这表示对该表的每个 INSERT 和 DELETE 都将不仅在表中插入和删除，而且会在其索引中插入和删除。如果对在索引中的列的值进行 UPDATE 操作，那么还更新了该索引。因此索引加快了检索过程的速度，但减慢了修改的速度。

- **现有索引中的列**：如果在 EMPNO 或 DEPTNO 上已经有了一个索引，那么创建另一个关于该组合的索引也许并不明智。但是，更改另一个索引以添加缺少的列也许可以起作用。但也不一定，因为索引中列的顺序也许会根据查询而有很大差异。例如，考虑以下查询：

```
SELECT   LASTNAME, SALARY
FROM     EMP
WHERE    EMPNO = '000010'
AND      DEPTNO >  'D01';
```

  在这种情况下，在索引中应该首先列出 EMPNO。然后列出 DEPTNO，从而允许 DB2 对第 1 列(EMPNO)执行直接索引查找，然后针对大于号(>)扫描第二列(DEPTNO)。

  而且，如果已经存在关于这两列的索引(一个关于 EMPNO，另一个关于 DEPTNO)，DB2 可以使用它们来满足该查询，因此创建另一个索引也许是没有必要的。

- **特定查询的重要性**：查询越重要，可能就越应该通过创建索引来进行调优。如果正在编写银行行长每天都运行的查询，那么应该确保它提供最佳性能。因此，为特定查询构建索引是很重要的。反之，职员的查询也许就没有必要看得那么重，所以也许应该利用现有索引来执行查询。当然，决定取决于应用程序对业务的重要性，而不只是用户的重要性。

索引设计涉及的内容比到目前为止我们上面讨论的要多得多。例如，也许要考虑索引包含列以实现完全索引访问。如果 SQL 查询要寻找的所有数据都包含在索引中，那么 DB2 也许只使用索引就可以满足该请求。请考虑我们前面的 SQL 语句。给定了关于 EMPNO 和 DEPTNO 的信息，我们要寻找 LASTNAME 和 SALARY。我们还从创建关于 EMPNO 和 DEPTNO 列的索引开始。如果我们在索引中还包含了 LASTNAME 和 SALARY，就不再需要访问 EMP 表，因为我们需要的所有数据都已经在索引中。该技术可以大大提高性能，因为它减少了 I/O 请求的数量。

请记住：使每个查询成为完全索引访问是不谨慎，甚至也是不可能的。应该谨慎使用该技术以便应用于特别棘手或重要的 SQL 语句。

7.8.3　配置合理的数据库配置参数

优化器在工作时，需要读取相关数据库配置参数，这些影响最直接、最重要的配置参数如下：

- **avg_appls**(平均应用程序)：此参数表示为数据库并发运行的应用程序平均数量。DB2 使用此信息来确定排序空间和缓冲池使用得有多么频繁，并确定查询能够使用的空间有多少。
- **sortheap**(排序堆)：排序堆是执行排序时可用的内存空间数量。如果排序需要的内存多于排序堆中的可用内存，那么部分排序数据将不得不分页到磁盘上(这会对性能造成严重的负面影响)。
- **locklist**(锁列表)：该参数表示 DB2 可用于存储各应用程序的锁定信息的内存数量。如果锁列表空间过小，那么 DB2 可能必须逐步升级(escalate)部分锁，以便为应用程序具有的所有锁腾出空间。
- **maxlocks**(最大锁列表百分比)：该参数控制整个锁列表空间中有百分之多少的空间可为应用程序所有。如果应用程序具有过多的开放锁，从而试图占用过多的内存，DB2 将升级部分锁以释放锁列表中的空间，这会影响并发性能。
- **num_freqvalues**(频繁值数)：RUNSTATS 实用工具使用频繁值数来控制 DB2 将在内存中保留多少使用频率最高的值。优化器使用该信息来确定 WHERE 子句中的一个谓词将消耗结果集的多少百分比。
- **num_quantiles**(分位数)：RUNSTATS 实用工具使用分位数来控制为列数据捕获多少分位。增加分位数将给予 DB2 关于数据库中数据分布情况的更多信息。
- **dbheap**(数据库堆)：数据库堆控制数据库对象信息的可用内存量。对象包括索引、缓冲池和表空间。事件监控器和日志缓冲区信息也存储在这里。
- **cpuspeed** (CPU 速度)：计算机的 CPU 速度。
- **buffpage 和缓冲池大小**：优化器可在优化数据中使用的缓冲池大小。增加或减少缓冲池大小会对访问计划产生显著影响。

7.8.4 选择合适的优化级别

对于混合的 OLTP/OLAP，使用 5 或 3 作为默认值；对于 OLTP，使用更低的级别；而对于 OLAP，则使用更高的级别。对于简单的 SELECT 或短的运行时查询(通常只需花不到 1s 就可以完成)，使用 1 或 0 也许比较合适。如果有很多的表，有很多相同列上的连接谓词，那么尝试级别 1 或 2。对于超过 30 秒才能完成的长时间运行的查询，或者要插入 UNION ALL VIEW(这是在 DB2 V8 FP4 中加进来的)，可以尝试使用级别 7。在大多数环境下都应该避免使用级别 9。一定要根据自己的业务类型和特点来选择合适的优化级别。

7.8.5 合理的存储 I/O 设计

优化器在工作时，需要读取相关硬件信息，这些信息是通过表空间的 transrate 和

overhead 两个参数体现的。

transrate=(1/传送速率)*1000/1024000*4096(假设用 4KB 页大小)
overhead=平均寻道时间+(((1/磁盘转速)*60*1000)/2)

而平均寻道时间、磁盘旋转速度和传送速率是由硬盘本身决定的。
所以我们必须做合理的存储 I/O 设计和选择转速更快的硬盘以使优化器更好地工作。

7.8.6　良好的应用程序设计和编码

在 DB2 数据库的应用开发中，主要执行的 SQL 语句有 INSERT、UPDATE、DELETE 和 SELECT 语句。其中 DML 操作(INSERT、UPDATE 和 DELETE)不需要返回结果，所以处理起来相对比较简单。而 SELECT 语句需要返回结果，在处理的时候相对比较复杂。优化器在处理 SELECT 语句时，通常假定应用程序必须检索由 SELECT 语句指定的所有行。此假设最适合于 OLTP 和批处理环境。但是，在“浏览”应用程序中，查询经常定义一个可能很大的结果集，但它们只检索前几行，通常只检索填满该屏幕所需的那么多行。

要提高这种应用程序的性能，可以按下列方式修改 SELECT 语句来影响优化器的优化策略：

- 使用 FOR UPDATE 子句来指定可由后续定位的 UPDATE 语句更新的列。
- 使用 FOR READ/FETCH ONLY 子句来使返回的列为只读的。
- 使用 OPTIMIZE FOR *n* ROWS 子句来给予检索整个结果集中前 *n* 行的优先级。
- 使用 FETCH FIRST *n* ROWS ONLY 子句来仅检索指定的几行。
- 使用 DECLARE CURSOR WITH HOLD 语句来每次检索一行。

如果使用 FOR UPDATE、FETCH FIRST *n* ROWS ONLY 或 OPTIMIZE FOR *n* ROWS 子句，或者如果声明游标为“滚动”，那么会影响行分块(BLOCKING)。

下面描述每个方法的性能优点。

FOR UPDATE 子句

FOR UPDATE 子句通过仅包含可由后续定位的 UPDATE 语句更新的列来限制结果集。如果不带列名指定 FOR UPDATE 子句，那么包括表或视图的全部可更新列。如果指定列名，那么每个名称必须是非限定的，并且必须标识表或视图的某一列。

在下列情况下，不能使用 FOR UPDATE 子句：

- 不能删除与 SELECT 语句关联的游标。
- 选择的列中至少有一列是系统目录表的不可更新列，并且没有在 FOR UPDATE 子句中排除掉。

FOR READ 或 FETCH ONLY 子句

FOR READ ONLY 子句或 FOR FETCH ONLY 子句确保返回只读结果。因为被定义为只读的视图上 SELECT 的结果表也是只读的，所以允许此子句但没有作用。

对于允许更新和删除的结果表，如果数据库管理器可以检索数据块而不是互斥锁定，那么指定 FOR READ ONLY 可能会提高 FETCH 操作的性能。不要对用于定位的 UPDATE 或 DELETE 语句的查询使用 FOR READ ONLY 子句。

OPTIMIZE FOR *n* ROWS 子句

OPTIMIZE FOR 子句声明只想检索结果的子集或优先检索前几行。优化器然后可以优先选择把检索前几行的响应时间减至最短的访问方案。另外，作为单个块发送到客户机的行数由 OPTIMIZE FOR 子句中的 n 值来限制。因此，OPTIMIZE FOR 子句既影响服务器从数据库检索合格行的方式，又影响将合格行返回到客户机的方式。

例如，假设定期查询职员表，查找具有最高工资的职员：

```
SELECT LASTNAME,FIRSTNAME,EMPNO,SALARY
        FROM EMPLOYEE        ORDER BY SALARY DESC
```

定义了一个基于 SALARY 列的降序索引。但是，由于职员是按职员号排序的，因此工资索引可能很难集群。为了尽量避免许多随机的同步 I/O，优化器将可能选择使用列表预取访问方法，此方法需要对合格的所有行的行标识排序。在将前几个合格行返回至应用程序前，此排序可能导致一个延迟。要防止此延迟，向语句添加 OPTIMIZE FOR 子句，如下所示：

```
SELECT LASTNAME,FIRSTNAME,EMPNO,SALARY        FROM EMPLOYEE
ORDER BY SALARY DESC          OPTIMIZE FOR 20 ROWS
```

在此情况下，优化器可能选择直接使用 SALARY 索引，因为只检索具有最高工资的 20 个职员。不管可以将多少行分块，将只把由每 20 行组成的块返回至客户机。

使用 OPTIMIZE FOR 子句，优化器优先选择可以避免大量操作或中断行流动(如排序)的访问方案。使用 OPTIMIZE FOR 1 ROW 最有可能影响访问路径。使用此子句可以有下列作用：

- 与组合内部表的连接顺序不太可能，因为它们需要临时表。
- 连接方法可以更改。嵌套循环连接是非常可能的选择，因为它具有低开销成本，并且通常在检索少量行时更有效率。
- 与 ORDER BY 子句匹配的索引更可能，因为对于 ORDER BY 不需要排序。
- 列表预取不太可能，因为此访问方法需要排序。

- 顺序预取不太可能，因为知道只需要少量的几行。
- 在连接查询中，在 ORDER BY 子句中包含列的表可能选作外部表，前提是该外部表上的索引提供 ORDER BY 子句所需的排序。

虽然 OPTIMIZE FOR 子句适用于所有优化级别，但它在优化级别 3 和更高级别下工作得最好，因为级别 3 以下的级别使用贪婪连接枚举方法。此方法有时会产生不能使它们自己很快检索前几行的多表连接的访问方案。

OPTIMIZE FOR 子句不阻止检索全部合格行。如果确实要检索全部合格行，那么总耗用时间可能大大高于优化器为整个答案集进行优化所需的时间。

如果已打包的应用程序使用调用级接口(DB2 CLI 或 ODBC)，可在 DB2CLI.INI 配置文件中使用 OPTIMIZEFORNROWS 关键字，让 DB2 CLI 自动将 OPTIMIZE FOR 子句追加至每条查询语句的末尾。

如果同时指定了 FETCH FIRST 子句和 OPTIMIZE FOR 子句，那么这两个值中较小的那个影响通信缓冲区大小。为了达到最优化，将这两个值看作互不相关的。

FETCH FIRST *n* ROWS ONLY 子句

FETCH FIRST *n* ROWS ONLY 子句设置可检索的最大行数。将结果表限制为只包含前几行可提高性能。无论结果集可能另外包含多少行，只检索 *n* 行。

如果同时指定了 FETCH FIRST 子句和 OPTIMIZE FOR 子句，那么这两个值中较小的那个影响通信缓冲区大小。为了达到优化目的，这两个值是互相独立的。

DECLARE CURSOR WITH HOLD 语句

当用包括 WITH HOLD 子句的 DECLARE CURSOR 语句声明游标时，在提交该事务时任何打开的游标仍然打开；并且释放所有锁定，除保护打开的 WITH HOLD 游标的当前游标位置的锁定外。如果回滚事务，那么关闭所有打开的游标且释放所有锁定以及释放 LOB 定位器。

可将 DB2 CLI 连接属性 SQL_ATTR_CURSOR_HOLD 用于 CLI 应用程序以实现同样的结果。如果有使用调用级接口(DB2 CLI 或 ODBC)的已打包的应用程序，那么在 DB2CLI.INI 配置文件中使用 CURSORHOLD 关键字，让 DB2 CLI 自动为每个已声明游标假设 WITH HOLD 子句。

7.9 本章小结

本章我们讨论了 DB2 数据库处理数据的核心引擎——DB2 优化器的功能、工作原理以及如何使其最有效地工作。优化器是数据库管理系统中产生执行计划的组件，本章主要

介绍了 DB2 优化器的工作原理，解读了执行计划中常见的操作以及它们的性能优劣，如索引扫描、全表扫描、合并连接、嵌套循环连接、哈希连接等。本章还介绍了对于不同的优化级别 DB2 采用的优化算法上面的差别，以及与 Oracle 类似的基于规则的优化概要文件。本章还就如何影响优化器以产生我们期望的执行计划给出了一些建议，如更新统计信息、构建索引、修改配置参数等。掌握了这些知识之后，我们就能够在设计配置数据库及应用开发的时候得心应手了。

第 8 章

统计信息更新与碎片整理

统计信息是 DB2 收集的关于数据库中表和索引等对象的相关信息，这些信息在收集之后被保存在数据库系统编目表中，当应用程序或 SQL 语句访问数据时，优化器需要根据这些统计信息来生成成本最低的访问计划。只有准确的统计信息才能让 DB2 数据库产生最优的数据访问计划，进而进行高效的数据访问。相反，如果数据库中只有过时的、不准确的统计信息，甚至没有相关的统计信息，那么数据的高效访问就无从谈起。所以统计信息的准确与否就显得非常重要。

随着数据的不断删除、插入和更新，数据页和索引页会变得越来越零散，数据页和索引页的物理存储顺序不再匹配其逻辑顺序，索引结构的层次会变得过大，这些都会导致数据页和索引页的读取变得效率低下。因此，需要根据数据更新的频繁程度适当地重新组织表和索引。

统计信息更新和表、索引碎片整理是 DBA 经常要做的例行工作，在本章中我们给大家讲解如下内容：

- 统计信息更新
- 自动统计信息更新
- 表、索引碎片整理
- 重新绑定应用程序包

8.1　统计信息更新

8.1.1　统计信息的重要性

DB2 优化器是 DB2 的“大脑”。而 DB2 优化器依靠存放在系统目录表空间中的数据库统计信息才能做出正确的判断，估算出某个特定查询的所有访问路径的成本。随后，DB2 优化器会选择成本最低的访问路径来获取数据。您可能会问，成本究竟表示什么呢？成本表示 DB2 优化器对执行语句所用时间的最优估计。在这里您需要注意，最常见的查询操作是 SELECT 语句，而 DML(比如 UPDATE)或 DDL(比如索引重建)也会执行查询操作，这同样需要通过基于成本的 DB2 优化器来进行分析。

数据库统计信息中的变化会影响 DB2 优化器对获取目标数据的访问路径成本的估算，从而可能选择不同的访问路径。因此，如果数据库统计信息误差过大，就有可能造成性能问题。

例如：一张有 100 万行数据的表，数据库统计信息却记录这个表只有一行数据，那么 DB2 根据统计信息就有可能选择全表扫描而不是索引去获取数据。

下面列举了统计信息收集的一些信息，这些信息可以帮助 DB2 优化器选择最优访问策略：

- 表中的页数和非空的页数。
- 表中发生行链接(overflow)的数量。
- 表中的行数。
- 有关单个列的统计信息，比如一列中唯一值的数量。
- 索引的聚合程度，即表中数据的存储顺序与某索引字段顺序的符合程度。
- 有关索引的统计信息，比如索引级别的数量和每个索引中叶子页的数量。
- 经常使用的列值的出现次数。
- 列值在列所有值中的分布状况。
- 用户定制函数(UDF)的成本估计。

除以上信息外，DB2 还可以收集下列信息：索引的聚合程度、索引中叶子页数目、溢出其原始页的表行数，以及表中已填充的页数和空页数。DBA 可以参考这些信息来决定何时重组表和索引。

DB2 数据库不可能在每次数据库添加、删除、修改数据后都更新数据库统计信息，这是因为过于频繁地更新统计信息有可能造成系统性能的巨大开销；此外，如果修改的数据数量有限，那么有可能统计信息的一些微误差不会造成 DB2 选择成本高昂的访问路径，这时也没有必要频繁更新数据库统计信息。出于上述两点原因，DB2 数据库中的统计信息并

不是动态更新的。不过，DB2 提供了 RUNSTATS 命令来手工更新数据库统计信息。当用户表中对大量数据修改后，用户可以考虑在表和索引上执行 RUNSTATS 命令，用最新的信息更新系统目录表中的统计信息。

在成功执行 RUNSTATS 命令之后，静态 SQL 查询并不会使用最新的数据库统计信息，这是因为静态 SQL 的访问策略在之前执行 BIND 时就已确定，而当时使用的统计信息有可能与现在的并不一致。这时候就需要重新绑定使用静态 SQL 的应用程序，这样查询优化器就可以根据数据库的最新统计信息来选择获取数据的最佳访问策略。但是，对于使用动态 SQL 的应用程序而言，没必要进行重新绑定，因为动态 SQL 语句的访问策略是根据统计信息在运行时动态生成的。

现在，几乎所有主流数据库都使用某种方法(在 Oracle 数据库中调用 dbms_stats 存储过程；在 Informix 数据库中用 update statistics；在 Sybase 数据库中用 update statistics)来更新系统目录统计信息，以便为优化器提供可能的最佳信息。您可以将优化器想象成汽车的 GPS 定位仪，它能够在由系统数据组成的莽莽深山里作行驶路径选择。目录统计信息的更新将为优化器提供最新、最准确的地图信息，以便能够获取最佳行驶路径。

8.1.2　如何更新统计信息

只有当进行显式请求时，对象的统计信息才会在系统目录表中被更新。有以下几种方法可以更新部分或全部统计信息：

- 使用 RUNSTATS(运行统计信息，run statistics)命令。
- 使用带有指定的统计信息收集选项的 LOAD。
- 对针对一组预先定义的系统目录视图进行操作的 SQL UPDATE 语句进行编码。
- 使用“reorgchk update statistics”命令。

当您不完全知道所有表名或表名实在太多而无法对每张表逐个更新策略时，进行 RUNSTATS 的最简单方法就是使用“db2 reorgchk update statistics”命令。脚本如下：

```
db2 -v connect to sample
db2 -v "select tbname, nleaf, nlevels, stats time from sysibm.sysindexes"
db2 -v reorgchk update statistics on table all
db2 -v "select tbname, nleaf, nlevels, stats time from sysibm.sysindexes"
db2 -v terminate
```

我们上面所选的示例不需要表名。这一命令对所有表执行 RUNSTATS 命令。

记住：
在批量数据加载后要运行 RUNSTATS 命令。

如果知道表名并且想避免对大量表执行 RUNSTATS 命令(因为这样做可能要花很长时间)，那么一次对一张表进行 RUNSTATS 更为可取。脚本如下：

```
db2 -v runstats on table  tabschema.tabname and indexes all
```

这个命令将收集该表及其所有索引(基本级别)的统计信息。

查看是否执行了 RUNSTATS 命令

要查看是否对数据库执行了 RUNSTATS 命令，一种快捷方法便是查询一些系统目录表。例如，对于上面的脚本，可以执行下面这条命令：

```
db2 -v "select tbname, nleaf, nlevels, stats_time from sysibm.sysindexes"
```

如果还未执行过 RUNSTATS 命令，您会看到 nleaf 和 nlevels 列为“-1”且 stats_time 列为“-”。如果已经执行了 RUNSTATS 命令，这些列将包含实际的数字，并且 stats_time 列将会包含时间戳记。如果您认为 stats_time 列显示的时间距离现在过久，就需要再次执行 RUNSTATS 命令。

可以查询系统目录表中的以下列，以确定是否在表和索引上执行了 RUNSTATS 命令：

- 如果对于某个表，SYSCAT.TABLES 视图的 CARD 列显示的值为－1，或者 stats_time 列显示的值为 NULL，那么表示没有对该表执行过 RUNSTATS 命令。
- 如果对于某个索引，SYSCAT.INDEXES 视图的 nleaf、nlevels 和 fullkeycard 列显示的值为－1，或者 stats_time 列显示的值为 NULL，那么表示还没有对该索引执行过 RUNSTATS 命令。

在表上执行 RUNSTATS 命令时，可以有两种用户访问选项：允许读访问(ALLOW READ ACCESS)和允许写访问(ALLOW WRITE ACCESS)。

在执行 RUNSTATS 命令时，如果加上 ALLOW READ ACCESS 选项，那么其他用户只能够以只读的方式访问该表，这个选项会影响应用的并行性，因为任何想要更改表的操作这时都会处于等待状态。可以选择在使用 ALLOW READ ACCESS 选项时，同时使用 TABLESAMPLE 选项，加上这个选项后只收集表的部分采样数据而不是所有数据。如果能合理选择采样数据大小，那么就可以在确保统计信息一致性的情况下，加快 RUNSTATS 的速度。

在执行 RUNSTATS 命令时，如果加上 ALLOW WRITE ACCESS 选项，那么其他用户可以读取或写入该表。默认情况下，RUNSTATS 命令使用的是 ALLOW WRITE ACCESS 选项。

分区数据库中的 RUNSTATS

在分区数据库环境中收集某个表的统计信息时，RUNSTATS 将只对从其中执行该实用程序的数据库分区执行操作。在此数据库分区中获得的结果将被推广到其他数据库分区。如果此数据库分区未包含该表的所需部分，那么该请求将被发送到数据库分区组中第一个包含所需数据的数据库分区。

8.1.3　统计信息更新示例

下面举一些例子来说明如何使用 RUNSTATS 收集统计信息。

注意：

在 RUNSTATS 语法中，必须使用全限定的表名 schema.table-name 和全限定的索引名 schema.index-name。您可以在所有列上，或者仅仅在某些列或列组(除了 LONG 和 LOB 列)上执行 RUNSTATS。如果没有指定特定列的子句，那么系统会使用默认的 ON ALL COLUMNS 子句。

例 8-1　收集所有列上的数据库统计信息。

```
RUNSTATS ON TABLE db2inst1.account ON ALL COLUMNS
```

这等同于：

```
RUNSTATS ON TABLE db2inst1.account
```

例 8-2　收集单个列上的数据库统计信息。

```
RUNSTATS ON TABLE db2inst1.account ON COLUMNS (acctno, acctname)
```

例 8-3　展示如何收集 key column(关键列)上的统计信息。短语“key column”表示构成表上所定义索引的列。如果没有索引存在，这条命令不会收集任何列的统计信息。

```
RUNSTATS ON TABLE db2inst1.account ON KEY COLUMNS
```

例 8-4　收集关键列上和非关键列上的数据库统计信息。

```
RUNSTATS ON TABLE db2inst1.account ON KEY COLUMNS AND COLUMNS (acctname)
```

例 8-5　收集表和索引上的数据库统计信息，不包含分布统计信息。

```
RUNSTATS ON TABLE db2inst1.account AND INDEXES ALL
```

例 8-6　收集表上的数据库统计信息以及索引上的详细统计信息，不包含分布统计

信息。

```
    RUNSTATS ON TABLE db2inst1.account AND DETAILED INDEXES ALL
```

例 8-7 只收集 3 个指定索引上的数据库统计信息(不收集表统计信息)。

```
    RUNSTATS ON TABLE db2inst1.account FOR INDEXES db2inst1.INX1,
db2inst1.INX2, db2inst1.INX3
```

例 8-8 只收集所有索引上的数据库统计信息。

```
    RUNSTATS ON TABLE db2inst1.account FOR INDEXES ALL
```

如果创建了新的索引，而在最后一次执行 RUNSTATS 以后还未修改相应的表，那么您可以只收集索引上的数据库统计信息。在创建索引时，还可以收集数据库统计信息。

包含统计信息配置文件的 RUNSTATS

现在，可以为 RUNSTATS 建立统计信息配置文件。统计信息配置文件是指一组选项，它预先定义了特定表上将要收集的统计信息。

当将命令参数“SET PROFILE”添加到 RUNSTATS 命令时，将在表描述符和系统目录中注册或存储统计信息配置文件。如果要更新统计信息配置文件，那么可以使用命令参数“UPDATE PROFILE”。DB2 中没有删除配置文件的选项。

那么这个配置文件有什么用途呢？

- LOAD 操作的时候使用选项 STATISTICS USE PROFILE 来收集统计信息。
- PROFILE 的 RUNSTATS 选项作为 AUTO RUNSTATS 的操作选项，关于 AUTO RUNSTATS 见 8.2 节。

下面的例子展示了如何使用这项功能：

例 8-9 只注册统计信息配置文件，不收集数据库统计信息。

```
    RUNSTATS ON TABLE db2inst1.account AND INDEXES ALL SET PROFILE ONLY
```

RUNSTATS 中的子句“SET PROFILE ONLY”指定不收集统计信息。

例 8-10 注册统计信息配置文件，并执行所存储统计信息配置文件的 RUNSTATS 命令选项来收集目录统计信息。

```
    RUNSTATS ON TABLE db2inst1.account AND INDEXES ALL SET PROFILE
```

例 8-11 仅修改现有的统计信息配置文件，不收集任何数据库统计信息。

```
    RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION AND INDEXES ALL UPDATE
PROFILE ONLY
```

例 8-12　修改现有的统计信息配置文件，并执行已更新的统计信息配置文件的 RUNSTATS 命令选项来收集数据库统计信息。

```
    RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION AND INDEXES ALL UPDATE
PROFILE
```

例 8-13　根据前面已注册的统计信息配置文件来查询 RUNSTATS 选项。

```
    SELECT STATISTICS PROFILE FROM SYSIBM.SYSTABLES WHERE NAME = 'DEPARTMENT'
AND CREATOR = 'DB2INST1'
```

例 8-14　使用前面已注册的统计信息配置文件收集数据库统计信息。

```
    RUNSTATS ON TABLE db2inst1.account USE PROFILE
```

带抽样的 RUNSTATS

随着数据库不断地快速增长，由于时间窗口、内存和 CPU 等约束的限制，通过全表扫描来收集数据库统计信息将会变得越来越困难。这时候可以考虑通过数据抽样，即只扫描表的数据子集来收集数据库统计信息。

如果查询试图预计总的趋势和模式，并且某一误差域(margin of error)内的近似答案足以监测这些趋势和模式，那么数据抽样或许是比全表扫描更好的选择。

使用 SAMPLED DETAILED 子句，这将通过对索引数据抽样来计算详细的索引统计信息。该子句的使用将减少为获得详细索引统计信息而执行的后台计算量和所需的时间，并且在大多数情况下，都会使数据足够精确。

注意：

在 DB2 V9 中，RUNSTATS 命令并不支持显式指定索引采样率。在 DB2 V10 以及之后的版本中，在 RUNSTATS 命令行中可以显式指定索引的采样率：

```
    Index Sampling Options

    |--INDEXSAMPLE--+-BERNOULLI-+--(--numeric-literal--)------------|

                    '-SYSTEM----'
```

以下是一些使用该子句的例子。

例 8-15　收集两个索引上的详细数据库统计信息，但对每个索引条目使用抽样来代替执行详细的计算。

```
    RUNSTATS ON TABLE db2inst1.account AND SAMPLED DETAILED INDEXES ALL
```

例 8-16　收集索引上的详细抽样统计信息和表的分布统计信息。

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION ON KEY
  COLUMNS AND SAMPLED DETAILED INDEXES ALL
```

DB2 提供了对表数据进行抽样的两种方法：行级的贝努里(Bernoulli)抽样和系统页级的抽样，这两种方法都可以在命令行中显式指定采样率。

```
Table Sampling Options

|--TABLESAMPLE--+-BERNOULLI-+--(--numeric-literal--)------------>

                 '-SYSTEM----'
```

带系统页级抽样的 RUNSTATS

系统页级抽样与行级抽样类似，只是抽样的对象是页面而不是行。以 P/100 的概率选择每一页，而以 1 - P/100 的概率拒绝页的选择。在所选中的每一页中，要选择所有的行。系统页级抽样优于全表扫描或贝努里(Bernoulli)抽样的地方是节省了 I/O。

抽样页也是预取的，所以该方法将比行级贝努里(Bernoulli)抽样更快。与不进行抽样相比，页级抽样极大地提高了性能。

RUNSTATS REPEATABLE 子句允许通过 RUNSTATS 语句生成相同的样本，只要表数据没有发生更改。为了指定该选项，用户还必须提供一个整数，以表示将用于生成样本的种子(seed)。通过使用相同的种子，可以生成相同的样本。

总之，统计信息的准确性取决于抽样率、数据倾斜(data skew)以及用于数据抽样的数据群集。

下面是一些使用贝努里(Bernoulli)行级和系统页级抽样的 RUNSTATS 例子。

例 8-17 收集统计信息，包含 10%行上的分布统计信息。

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION TABLESAMPLE BERNOULLI (10)
```

例 8-18 为了控制收集统计信息的样本集，以及可以重复使用相同的样本集，需要使用下列语句：

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION TABLESAMPLE
   BERNOULLI (10) REPEATABLE (1024)
```

例 8-19 收集 10%的数据页上的索引统计信息和表统计信息。请注意，只对表数据页进行抽样，而不是索引页。在本例中，10%的表数据页用于表统计信息的收集，而对于索引统计信息，将使用所有的索引页。

```
R6UNSTATS ON TABLE db2inst1.account AND INDEXES ALL TABLESAMPLE SYSTEM (10)
```

8.1.4　LIKE STATISTICS 统计信息更新

我们在应用中经常使用类似%abc%的查询，在某些列上指定 LIKE 谓词：

```
    SELECT *  FROM  employee WHERE  firstname LIKE '%abc%'
```

当在 RUNSTATS 中指定 LIKE STATISTICS 子句时，将收集附加的列统计信息。查询优化器用它们来提高“column LIKE '%abc'”和“column LIKE '%abc%'”类型谓词的选择性估计。它通常有利于优化器了解某些关于列的子元素结构的基本统计信息。这些统计信息存储在 SYSIBM.SYSCOLUMNS 表的 SUB_DELIM_LENGTH 和 SUB_COUNT 列中。当执行带有 LIKE STATISTICS 子句的 RUNSTATS 时，将对类型为 CHAR 和 VARCHAR，并且具有单字节字符集(SBCS)的代码页属性 FOR BIT DATA 或 UTF-8 的列收集下列统计信息：

- SUB_COUNT：子元素的平均数。
- SUB_DELIM_LENGTH：分隔每个子元素的每个定界符的平均长度。

下面的例子说明了如何使用 RUNSTATS 收集包含 LIKE 统计信息的数据库统计信息。

例 8-20　收集所有列上的数据库统计信息并指定 VARCHAR 列上的 LIKE 统计信息。

```
   RUNSTATS ON TABLE db2inst1.account ON ALL COLUMNS and COLUMNS (acctname LIKE
STATISTICS)
```

DB2_LIKE_VARCHAR 注册变量影响优化器处理以下格式谓词的方式：

```
   COLUMN LIKE '%xxxxxx%'
```

其中 xxxxxx 代表任何字符串，也就是搜索值从%字符开始的任何 LIKE 谓词(可能以%字符结束，也可能不以该字符结束)。将这些称为“通配符 LIKE 谓词”。对于所有谓词，优化器必须估计有多少行与该谓词匹配。对于通配符 LIKE 谓词，优化器假设要匹配的 COLUMN 包含并置在一起的一系列元素，并且基于字符串长度估计每个元素的长度，不包括前导和结尾%字符。

要检查子元素统计信息的值，查询 SYSIBM.SYSCOLUMNS。例如：

```
   select substr(NAME,1,16), SUB COUNT, SUB DELIM LENGTH          from
sysibm.syscolumns where tbname = 'EMPLOYEE'
```

注意：

如果使用了 LIKE STATISTICS 子句，那么 RUNSTATS 可能花较长时间。例如，如果未使用 DETAILED 和 DISTRIBUTION 选项，那么 RUNSTATS 在具有 5 个字符列的表上运行时间会增加 15%到 40%，甚至更长时间。如果指定 DETAILED 或 DISTRIBUTION 选项，那么相应也会额外增加一些开销。所以我们在应用时一定要评估给我们带来的性能提

高和相应延长的 RUNSTATS 时间之间的开销是否相当。通常，如果类似“select colname from t1 where colname like '%ABC%'” 这样的 SQL 语句执行得非常频繁而且该查询比较重要，那么建议对该列做 LIKE STATISTICS 统计信息更新。

8.1.5 列组统计信息更新

我们在“第 7 章：DB2 优化器”中给大家讲过，DB2 优化器在为 SQL 语句执行最优计划时，成本评估产生 3 种度量标准：

- SELECTIVITY：表示有多少行可以通过谓词被选择出来，大小介于 0.0~1.0，0 表示没有行被选择出来。如果没有统计信息，estimator 会使用默认的 SELECTIVITY 值，这个值根据谓词的不同而不同。比如 '='的 SELECTIVITY 小于'<'。如果有统计信息，比如对于 last_name = 'Smith'，estimator 使用 last_name 列的 distinct 值的倒数(是指表中所有 last_name 的 distinct 值)作为 SELECTIVITY 值。
- CARDINALITY：表示行集的行数。
- COST：COST 表现了磁盘 I/O，CPU usage 资源单位的使用成本单位(timeron)。

列组(Column Group)统计信息将获得一组列的不同值组合的数目。通常，DB2 优化器可用的基本统计信息不检测数据相关性。列组的使用将给多个谓词的联合选择提供更准确的估计。优化器凭借精确的基数估计(Estimated Cardinality)值来准确计算出每一个待定查询访问计划的成本。基数估计是这样一种过程：在应用了谓词或执行了聚集之后，优化器使用统计信息确定部分查询结果的大小。对于访问计划的每个操作符，优化器将估计该操作符的基数输出。应用一个或更多个谓词可以减少输出流基数。

在计算谓词对于基数估计值的组合过滤效果时，通常会假设这些谓词彼此之间是独立的。但在统计方面，这些谓词可以彼此关联。单独地处理多个谓词通常会导致优化器低估基数值。而基数值的低估又会导致优化器选择次优的访问计划。

优化器会考虑使用多列统计信息来检测统计关联，并更加准确地估计多个谓词组合的过滤效果。本节描述了优化器如何利用多列统计信息来检测统计关联，并更加准确地估算多个等式谓词对应用了至少两个本地 IN、OR 和等式谓词的 SQL 语句的组合过滤效果，以及应用了某种等级的 OR 谓词的 SQL 语句的过滤效果。利用 DB2 列组统计信息，优化器可以在多个谓词相关联时确定更好的查询访问计划并提高查询性能。

多个本地等式和本地 IN 谓词的统计关联

如果 SQL 语句的 WHERE 子句使用了多个谓词：

```
C1=? AND C2 IN ( ?, ?, ? )
```

并且收集了(C1，C2)的多列统计信息的话，那么优化器就会试着检测这些谓词间的统

计关联以提高基数估计值。但以下谓词除外：

- 带有 IN 或 OR 操作符的连接谓词。
- 带有不等式、LIKE 或 IS NULL 操作符的本地谓词。
- 带有子查询的谓词。

“C1=?”谓词就是本地等式谓词的一个例子。本地等式谓词是应用于单个表的等式谓词，描述如下：

```
COLUMN = literal
```

其中的 literal 可以是以下任一内容：

- 常量值
- 参数标记或主变量
- 专用寄存器(例如 CURRENT DATE)

“C2 IN (?, ?, ?)”谓词则是本地 IN 谓词的一个例子。本地 IN 谓词是应用于同一表格——与本地谓词应用的表格相同——的等式谓词，描述如下：

```
COLUMN IN ( <VALUE LIST> )
```

其中<VALUE LIST>是以逗号隔开的一个或多个上述(在本地等式谓词中)literal 的列表。

相当于 IN 谓词的 OR 谓词可以代替 IN 谓词在 SQL 语句中指定，而且优化器将会在说明统计关联时按相同的方式处理之，也就是说：

```
COL IN  ( literal_1, literal_2, ..., literal_n )
```

相当于

```
 COL=literal_1 OR COL=literal_2 OR ... OR COL=literal_n
```

下面是优化器为本地 IN、OR 和等式谓词检索统计关联的例子：

```
    COL 1 IN ( <VALUE LIST> ) AND COL 2=literal AND COL 3=literal
    (COL 1=literal 1 OR COL 1=literal 2 OR ... OR COL 1=liternal n) AND
COL 2=literal AND ... AND COL m=literal
    COL 1 IN ( <VALUE LIST> ) AND COL 2 IN ( <VALUE LIST> ) AND ... AND COL m
IN ( <VALUE LIST> )
    (COL 1=literal 1 OR COL 1=literal 2) AND (COL 2=literal 1 OR
COL 2=literal 2) AND ... AND (COL m=literal 1 OR COL M=literal 2)
    COL 1 IN ( <VALUE LIST> ) AND ... And COL m IN ( <VALUE LIST> ) AND
COL 1 2=literal AND ... AND COL 1 k=literal
    (COL_1=literal_1 OR COL_1=literal_2) AND COL_2=literal AND COL_3=literal
```

```
    (COL 1=literal 1 OR COL 1=literal 2) AND (COL 2=literal 1 OR
COL_2=literal_2) AND COL_3=literal
```

下面这些是优化器不会考虑为其检测统计关联的谓词的例子：

```
    (COL 1=literal AND COL 2=literal) OR (COL 1=literal AND COL 2=literal AND
COL 3=literal)
    (COL 1=literal AND COL 2=literal) OR (COL 1=literal AND COL 2=literal)) AND
COL 3=literal
    ( COL_1 IN ( <VALUE LIST> ) OR (COL_2 IN ( <VALUE LIST> ) ) AND COL_3=literal
```

下面举几个列组统计信息更新的示例：

例 8-21　C1 IN (<VALUE LIST>) AND C2 = literal。

考虑对 SAMPLE 数据库的 EMPLOYEE 表执行如下查询：

```
    SELECT FIRSTNME, LASTNAME, JOB, WORKDEPT, SALARY        FROM EMPLOYEE
WHERE JOB IN ('CLERK', 'SALESREP') AND     WORKDEPT = 'A00'      ORDER BY JOB,
SALARY
```

该查询从 EMPLOYEE 表返回 4 条记录：

```
    FIRSTNME     LASTNAME        JOB      WORKDEPT SALARY
    ------------ --------------- --------- --------- -----------
    GREG         ORLANDO         CLERK    A00         39250.00
    SEAN         O'CONNELL       CLERK    A00         49250.00
    DIAN         HEMMINGER       SALESREP A00         46500.00
    VINCENZO     LUCCHESSI       SALESREP A00         66500.00
    4 record(s) selected.
```

SAMPLE 在创建最初，还没有在表上收集统计信息。为了保证执行计划的准确性，需要更新表 EMPLOYEE 的统计信息。下面的 RUNSTATS 命令在 EMPLOYEE 表的每一列上收集了统计信息，包括分布统计信息和所有在 EMPLOYEE 表中定义的索引上的详细统计信息(如果存在的话)：

```
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE   WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

下面我们使用 db2expln 来查看该 SQL 语句的执行计划：

```
    $db2expln -d sample -q "SELECT FIRSTNME, LASTNAME, JOB, WORKDEPT, SALARY
FROM EMPLOYEE        WHEAND     WORKDEPT = 'A00'      ORDER BY JOB, SALARY" -t
    Estimated Cost = 7.669430
    Estimated Cardinality = 1.190476
    Access Table Name = ORACLE.EMP1  ID = 3,12
    |  #Columns = 4
```

```
    |  Avoid Locking Committed Data
    |  Evaluate Block/Data Predicates Before Locking Committed Row
    |  Relation Scan
    ..............................略..........................
```

访问计划如上所示，它的估计基数为 1.190476，该值与实际返回的 4 行结果不符。这是因为优化器假定两个谓词是独立的，因为相关的索引或列组统计信息不存在。我们可以使用 RUNSTATS 命令在(JOB,WORKDEPT)上收集列组统计信息，以此为优化器提供适当的信息来检测两个列之间的统计关联(如果存在的话)：

```
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE ON ALL COLUMNS  AND COLUMNS
((JOB,WORKDEPT)) WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

重复上面的步骤，再次解释查询，然后生成查询访问计划，优化器会计算出一个更好的基数估计值，因为它在两个列上收集了列组统计信息：

```
     $db2expln -d sample -q "SELECT FIRSTNME, LASTNAME, JOB, WORKDEPT, SALARY
FROM EMPLOYEE       WHERE JOB IN ('CLERK', 'SALESREP') AND     WORKDEPT = 'A00'
ORDER BY JOB, SALARY" -t
    Estimated Cost = 118.487167
    Estimated Cardinality = 5.000000
    Access Table Name = ORACLE.EMPLOYEE  ID = 2,6
    |  #Columns = 4
    |  Volatile Cardinality
    |  Avoid Locking Committed Data
    |  Evaluate Block/Data Predicates Before Locking Committed Row
    |  Relation Scan
    ............................略.................................
```

由于列组统计信息是均匀分布的统计信息，而实际数据分布是不均匀的，因此基数估计值比实际值 4 高了一些。您可能已经注意到了，访问计划本身的评估成本(Estimated Cost)并未随着基数估计值的增大而显著改变。这是因为本例中描述的例子都很简单，数据量非常小(只有 42 条记录)。如果语句涉及更大的表和两个或更多个表的连接的话，查询访问计划就很可能会因为基数估计值的提高而显著改变。

例 8-22　C1 IN (<VALUE LIST>) AND C2 IN (<VALUE LIST>)。

这个示例解释说明了在两个 IN 谓词上的列组统计信息的效果。考虑以下检索某一部门的经理和设计人员的奖金和薪水的查询：

```
    SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS, SALARY       FROM EMPLOYEE
WHERE WORKDEPT IN ('D11','D21') AND          JOB IN ('MANAGER','DESIGNER')
ORDER BY WORKDEPT, SALARY
```

该查询从 EMPLOYEE 表返回 12 条记录：

```
FIRSTNME     LASTNAME        WORKDEPT JOB      BONUS       SALARY
------------ --------------- -------- -------- ----------- -----------
MASATOSHI    YOSHIMURA       D11      DESIGNER 500.00      44680.00
JENNIFER     LUTZ            D11      DESIGNER 600.00      49840.00
JAMES        WALKER          D11      DESIGNER 400.00      50450.00
MARILYN      SCOUTTEN        D11      DESIGNER 500.00      51340.00
BRUCE        ADAMSON         D11      DESIGNER 500.00      55280.00
DAVID        BROWN           D11      DESIGNER 600.00      57740.00
ELIZABETH    PIANKA          D11      DESIGNER 400.00      62250.00
KIYOSHI      YAMAMOTO        D11      DESIGNER 500.00      64680.00
WILLIAM      JONES           D11      DESIGNER 400.00      68270.00
REBA         JOHN            D11      DESIGNER 600.00      69840.00
IRVING       STERN           D11      MANAGER  500.00      72250.00
EVA          PULASKI         D21      MANAGER  700.00      96170.00
  12 record(s) selected.
```

首先，在没有获得(JOB，WORKDEPT)列组统计信息的情况下检查访问查询计划和基数估计值。可以按如下方式在 EMPLOYEE 表上执行另一个 RUNSTATS 命令来完成该检查：

```
RUNSTATS ON TABLE DB2INST1.EMPLOYEE   WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

由于前面收集的统计信息被最新的 RUNSTATS 命令清除了，因此先前收集的列组统计信息不复存在。再次运行 db2expln 来查看该 SQL 语句的执行计划，以此来检查优化器估计的基数：

```
$db2expln -d sample -q "SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS,
SALARY       FROM EMPLOYEE      WHERE WORKDEPT IN ('D11','D21') AND
JOB IN ('MANAGER','DESIGNER')       ORDER BY WORKDEPT, SALARY " -t
    Estimated Cost = 7.668932
    Estimated Cardinality = 7.285715
    Access Table Name = ORACLE.EMP1  ID = 3,12
    |  #Columns = 6
    |  Avoid Locking Committed Data
    |  Evaluate Block/Data Predicates Before Locking Committed Row
    |  Relation Scan
    |  |  Prefetch: Eligible
    |  Lock Intents
    |  |  Table: Intent Share
    ........................略................................................
```

基数估计值 7.285715 与实际返回的 12 行结果不符。和例 8-21 一样，在(JOB，WORKDEPT)上收集列组统计信息会在计算两个 IN 谓词的组合过滤效果时为优化器提供必要的信息，用以说明统计关联：

```
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE    ON ALL COLUMNS AND COLUMNS
((JOB,WORKDEPT))    WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

重复上面的步骤，再次解释查询，在生成查询访问计划之后，优化器会计算出一个更好的、与实际结果很接近的基数估计值 11.200000：

```
    $db2expln -d sample -q "SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS,
SALARY       FROM EMPLOYEE      WHERE WORKDEPT IN ('D11','D21') AND
JOB IN ('MANAGER','DESIGNER')       ORDER BY WORKDEPT, SALARY " -t
    Estimated Cost = 117.296455
    Estimated Cardinality = 11.200000
    Access Table Name = ORACLE.EMPLOYEE  ID = 2,6
    |  #Columns = 6
    |  Volatile Cardinality
    |  Avoid Locking Committed Data
    ..............................略..............................
```

例 8-23　C1 IN (<VALUE LIST>) AND C2 IN (<VALUE LIST>) AND C3=literal。

在这个例子中，将向例 8-22 的查询中添加第 3 个谓词，以确定有哪些职工得到了$500 的奖金：

```
    SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS, SALARY  FROM EMPLOYEE
WHERE WORKDEPT IN ('D11','D21') AND       JOB IN ('MANAGER','DESIGNER') AND
BONUS = 500  ORDER BY WORKDEPT, SALARY
```

该查询从 EMPLOYEE 表返回 5 条记录：

```
    FIRSTNME     LASTNAME        WORKDEPT JOB       BONUS      SALARY
    ------------ --------------- -------- --------- ---------- -----------
    MASATOSHI    YOSHIMURA       D11      DESIGNER  500.00     44680.00
    MARILYN      SCOUTTEN        D11      DESIGNER  500.00     51340.00
    BRUCE        ADAMSON         D11      DESIGNER  500.00     55280.00
    KIYOSHI      YAMAMOTO        D11      DESIGNER  500.00     64680.00
    IRVING       STERN           D11      MANAGER   500.00     72250.00
      5 record(s) selected.
```

如果使用如下代码：

```
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE
```

```
    WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

在没有列组统计信息的情况下再次收集统计信息，优化器会选择类似如下所示的查询访问计划，它的基数估计值为 2.428572：

```
    $db2expln -d sample -q "SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS, 
SALARY  FROM EMPLOYEE  WHERE WORKDEPT IN ('D11','D21') AND   JOB IN 
('MANAGER','DESIGNER') AND        BONUS = 500  ORDER BY WORKDEPT, SALARY" -t
    Estimated Cost = 7.683424
    Estimated Cardinality = 2.428572
    Access Table Name = ORACLE.EMP1  ID = 3,12
    |  #Columns = 5
    |  Avoid Locking Committed Data
    |  Evaluate Block/Data Predicates Before Locking Committed Row
    |  Relation Scan
    |  |  Prefetch: Eligible
    |  Lock Intents..............................略..................................
```

WHERE 子句中应用了 3 个谓词，如果假设它们是独立的，那么会导致优化器低估基数。为了解释说明优化器如何使用索引统计信息以及列组统计信息来检测统计关联，创建带有在谓词中引用的 3 个列(JOB，WORKDEPT，BONUS)的索引并收集统计信息：

```
    CREATE INDEX JOB DEPT BONUS ON EMPLOYEE (JOB,WORKDEPT,BONUS)
    -- The RUNSTATS command provides the option to collect statistics on a set of
    -- indexes only, without affecting the statistics previously collected.
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE FOR DETAILED INDEXES 
DB2INST1.JOB_DEPT_BONUS
```

然后优化器会使用新创建的索引和在其上收集的统计信息来更正查询访问计划的基数估计值：

```
    Estimated Cost = 7.668263
    Estimated Cardinality = 5.250000-------这个值和实际返回的 5 行相近
    Table Constructor
    |  2-Row(s)
    Nested Loop Join
    |  Access Table Name = ORACLE.EMP1  ID = 3,12
    |  |  Index Scan:  Name = ORACLE.JOB_DEPT_BONUS  ID = 1
    |  |  |  Regular Index (Not Clustered)
    |  |  |  Index Columns:
    |  |  |  |  1: JOB (Ascending)
    |  |  |  |  2: WORKDEPT (Ascending)
    |  |  |  |  3: BONUS (Ascending)
```

```
    |  |  #Columns = 5..............................略..............................
```

例 8-24　C1=literal OR C1=literal2) AND (C2=literal OR C2=literal2) AND C3=literal。这个例子与例 8-23 等效，它使用了等效的 OR 谓词来代替 IN 谓词：

```
    SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS, SALARY        FROM EMPLOYEE
WHERE  (WORKDEPT = 'D11' OR WORKDEPT = 'D21') AND          (JOB = 'MANAGER' OR
JOB = 'DESIGNER') AND           BONUS = 500      ORDER BY WORKDEPT, SALARY
```

该查询返回的结果与例 8-23 相同。这个示例解释说明了部分统计信息对于优化器估计基数的能力的影响。删除例 8-23 中创建的索引，并只使用(JOB，WORKDEPT)上的列组统计信息再次收集统计信息：

```
    DROP INDEX JOB DEPT BONUS
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE   ON ALL COLUMNS AND COLUMNS
        ((JOB,WORKDEPT))   WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

使用在合适的 IN、OR 和等式谓词引用的列的子集上收集的列组统计信息，优化器估计出了一个更接近于真实结果的基数。但如果列组统计信息是在全部 3 个列上收集的话，该估计值的精确度要低于例 8-23 中所示的值：

```
    Estimated Cost = 7.683424
    Estimated Cardinality = 2.428572
    Access Table Name = ORACLE.EMP1  ID = 3,12
    |  #Columns = 5
    |  Avoid Locking Committed Data
    |  Evaluate Block/Data Predicates Before Locking Committed Row
    |  Relation Scan
    |  |  Prefetch: Eligible
    |  Lock Intents
    ..............................略..............................
```

优化器使用了在(JOB，WORKDEPT)上收集的列组统计信息来说明两个 OR 谓词间的统计关联，但不包括列组中的 BONUS，它认为 BONUS=500 谓词独立于那两个 OR 谓词，结果导致稍稍低估了最终的基数。

注意：

如果使用可视化解释或 db2exfmt 输出的 Optimized Statement 部分，那么您会注意到该 SQL 语句的 OR 谓词被转换成与它们等效的 IN 谓词：

```
    Optimized Statement:
    -------------------
```

```
    SELECT Q5.FIRSTNME AS "FIRSTNME", Q5.LASTNAME AS "LASTNAME", Q5.WORKDEPT
AS"WORKDEPT", Q5.JOB AS "JOB", +0000500.00 AS "BONUS", Q5.SALARY AS
         "SALARY"FROM DB2INST1.EMPLOYEE AS Q5
    WHERE (Q5.BONUS = +0000500.00) AND Q5.JOB IN ('MANAGER ', 'DESIGNER') AND
         Q5.WORKDEPT IN ('D11', 'D21') ORDER BY Q5.WORKDEPT, Q5.SALARY
```

在全部 3 列上收集列组统计信息会导致与例 8-23 相同的基数估计值。在这种情况下，您仍然在前面的(JOB，WORKDEPT)上收集列组统计信息，并包括整个 3 列(JOB，WORKDEPT，BONUS)：

```
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE   ON ALL COLUMNS AND COLUMNS
       ((JOB,WORKDEPT), (JOB,WORKDEPT,BONUS))  WITH DISTRIBUTION AND DETAILED
       INDEXES ALL
```

您可以在相同的列集合间收集到一个或更多的列组统计信息。收集这些统计信息后，生成的查询访问计划与例 8-23 中最后的计划相同。至于这种情况的验证就留给读者自己做练习。

例 8-25 (C1=LITERAL1 AND C2=LITERAL2)OR(C1=LITERAL3 AND C2=LITERAL4)。

这个例子解释说明了列组统计信息对合格的 OR 谓词的影响。考虑在 EMPLOYEE 表上执行如下查询：

```
    SELECT FIRSTNME, LASTNAME, WORKDEPT, JOB, BONUS, SALARY  FROM EMPLOYEE
WHERE(WORKDEPT='E21' AND JOB='FIELDREP')OR(WORKDEPT='D21'AND JOB='MANAGER')
ORDER BY WORKDEPT, SALARY
```

该查询从 EMPLOYEE 表返回 6 条记录：

```
    FIRSTNME     LASTNAME        WORKDEPT JOB      BONUS       SALARY
    ----------- --------------- -------- -------- ----------- -----------
    EVA          PULASKI         D21      MANAGER  700.00      96170.00
    ROY          ALONZO          E21      FIELDREP 500.00      31840.00
    HELENA       WONG            E21      FIELDREP 500.00      35370.00
    RAMLAL       MEHTA           E21      FIELDREP 400.00      39950.00
    JASON        GOUNOT          E21      FIELDREP 500.00      43840.00
    WING         LEE             E21      FIELDREP 500.00      45370.00
     6 record(s) selected.
```

如果使用如下代码：

```
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE  WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

在没有列组统计信息的情况下再次收集统计信息的话，优化器会选择类似如下所示的查询访问计划，它的基数估计值为 1.880952：

```
    Estimated Cost = 7.718598
    Estimated Cardinality = 1.880952
    Access Table Name = ORACLE.EMP1  ID = 3,12
    |  #Columns = 6
    |  Avoid Locking Committed Data
    |  Evaluate Block/Data Predicates Before Locking Committed Row
    |  Relation Scan
    |  |  Prefetch: Eligible
    |  Lock Intents
    |  |  Table: Intent Share..........................略....................................
```

在(JOB，WORKDEPT)上收集列组统计信息能够让优化器更好地估计 OR 谓词的过滤效果，因为 OR 谓词的每一个子项都在 JOB 和 WORKDEPT 列上应用了一组本地等式谓词。确定合适的 RUNSTATS 语句来收集列组统计信息：

```
    RUNSTATS ON TABLE DB2INST1.EMPLOYEE   ON ALL COLUMNS AND COLUMNS
((JOB,WORKDEPT))  WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

收集了列组统计信息之后，优化器会选择类似如下所示的查询访问计划，它的基数估计值提高了，与实际的结果(6 行)很接近：

```
    Estimated Cost = 128.489594
    Estimated Cardinality = 5.600000
    Access Table Name = ORACLE.EMPLOYEE  ID = 2,6
    |  #Columns = 6
    |  Volatile Cardinality
    |  Avoid Locking Committed Data
    |  Evaluate Block/Data Predicates Before Locking Committed Row
    |  Relation Scan
    |  |  Prefetch: Eligible
    ..............................略..........................................
```

上面我们给大家讲解了一些列组更新的示例，优化器凭借精确的基数估计值来准确计算出每一个待定查询访问计划的成本。您可以利用 DB2 中的列组统计信息的扩展用途来为优化器提供更多的信息，从而使优化器更好地估计基数、选择最佳的查询访问计划。列组统计信息对于处理复杂的 OLAP 查询来说尤其重要。

8.1.6　分布统计信息更新

当您确定表中数据分布不均匀时，可以运行包含 WITH DISTRIBUTION 子句的 RUNSTATS 命令。系统目录表中的统计信息通常包含关于表中最高值和最低值的信息，而优化器假定数据值是在两个端点值之间均匀分布的。然而，如果数据值彼此之间差异较大，

或者群集在某些点上，或者是碰到许多重复的数据值，那么优化器就无法选择最佳的访问路径，除非收集了分布统计信息。使用 WITH DISTRIBUTION 子句还可以帮助查询处理没有参数标记(parameter marker)或主机变量的谓词，因为优化器仍然不知道运行时的值是有许多行，还是只有少数行。

可以收集两种数据分布统计信息：

- 频率统计信息：这些统计信息提供关于 *num_freqvalues* 数据库配置参数的值所指定级别的具有最高重复值和次高重复值的列和数据值的信息。其默认值是 10，建议将这个值设置在 10 到 100 之间。如果将 *num_freqvalues* 设置为零，那么不保留任何频率值的统计信息。还可以将 *num_freqvalues* 设置为每个表、统计信息视图和特定列的 RUNSTATS 选项。
- 分位数统计信息：这些统计信息提供关于与其他值相比如何分布数据值的信息。称为 K 分位数，这些统计信息表示值 V，至少 K 个值位于该值或该值以下。可以通过按升序排序值来计算 K 分位数。K 分位数值是从范围的低端起第 K 个位置中的值。

要指定应该将列数据值分组成的部分数，将 *num_quantiles* 数据库配置参数设置为 2 到 32 767 之间的某个值。默认值为 20，它确保对任何相等、小于或大于谓词的优化器估计误差最大为正或负 2.5%，而对任何 BETWEEN 谓词的最大误差为正或负 5%。要禁用分位数统计信息收集，将 *num_quantiles* 设置为 0 或 1。可以对每个表或统计信息视图以及特定列设置 *num_quantiles*。

提示：

如果没有在 RUNSTATS 命令的列或表级别上指定 *num_freqvalues* 和 *num_quantiles*，那么 *num_freqvalues* 的值将从 *num_freqvalues* 数据库配置参数中获取，而 *num_quantiles* 的值将从 *num_quantiles* 数据库配置参数中获取。如果指定较大的 *num_freqvalues* 和 *num_quantiles* 值，那么执行 RUNSTATS 时将需要更多的 CPU 资源和内存。通过 *stat_heap_sz* 数据库配置参数来指定 CPU 资源和内存量。

可以为单个列或一组列修改频率和分位数统计信息的精确度。提高分布统计信息的精确度将导致更大的 CPU 和内存消耗，并占用更多的系统目录表空间。对于这些分布统计信息，只考虑对于拥有选择谓词的最重要的查询而言最为重要的列。

当出现下列任何一种条件时，RUNSTATS 将不收集分布统计信息：

- 当将 *num_freqvalues* 配置参数设置为零(0)，以及当将 *num_quantiles* 数据库配置参数设置为零(0)或 1 时。
- 当每个数据值是唯一的时候。

- 当该列是 LONG、LOB 或结构化列时。
- 当列中只有一个非空值时。
- 声明的临时表。

何时收集分布统计信息

要决定是否应创建和更新给定表或统计信息视图的分布统计信息，考虑以下两个因素：

- 应用程序是否使用静态或动态 SQL 和 XQuery 语句。分布统计信息对不使用主变量的动态查询和静态查询最有用。当使用具有主变量的查询时，优化器只能有限地利用分布统计信息。
- 列中的数据是否是均匀分布的。如果该表中至少有一列的数据分布非常“不均匀”，并且该列频繁出现在等式或范围谓词中，也就是类似如下所示的子句中，那么考虑收集分布统计信息。

```
WHERE C1 = KEY;
WHERE C1 IN (KEY1, KEY2, KEY3);
WHERE (C1 = KEY1) OR (C1 = KEY2) OR (C1 = KEY3);
WHERE C1 <= KEY;
WHERE C1 BETWEEN KEY1 AND KEY2;
```

可能发生两种类型的不均匀数据分布(可能一起发生)：

- 可能按一个或多个子间隔集群数据，而不是均匀地在最高和最低数据值之间分布。考虑以下列，其中的数据在范围(5, 10)内集群：

  C1

  0.0

  5.1

  6.3

  7.1

  8.2

  8.4

  8.5

  9.1

  93.6

100.0

分位数统计信息可以帮助优化器处理这种类型的数据不均匀分布。

要帮助确定是否不是均匀分布的列数据，执行诸如以下所示的查询：

```
SELECT C1, COUNT(*) AS OCCURRENCES FROM T1 GROUP BY C1 ORDER BY OCCURRENCES DESC;
```

- 重复数据值可能经常出现。考虑使用表 8-1 所示频率分布数据的列。

表 8-1　数据及其频率分布

数　据　值	频　　率
20	5
30	10
40	10
50	25
60	25
70	20
80	5

要帮助优化器处理重复值，可以创建分位数和高频值统计信息。

要指定哪个级别的统计精度

要确定存储分布统计信息使用的精度，指定数据库配置参数 *num_quantiles* 和 *num_freqvalues*。还可以指定这些参数作为收集表或列的统计信息时的 RUNSTATS 选项。这些值设置得越大，RUNSTATS 创建和更新分布统计信息时使用的精度越高。但是，精度越高，在 RUNSTATS 执行期间和在目录表中所需要的存储器中需要使用的资源就越多。

对于大多数数据库，为 *num_freqvalues* 数据库配置参数指定 10 到 100 之间的数。理论上，应创建高频值统计信息，以便其余值的频率可近似等于最高频值的频率，或者相比之下可以忽略不计。数据库管理器收集的数目可能低于此数，因为只对出现多次的数据值收集这些统计信息。如果需要只收集分位数统计信息，那么将 *num_freqvalues* 设置为 0。

要设置分位数的数目，指定 20 到 50 之间的数作为 *num_quantiles* 数据库配置参数的值。确定分位数数目的经验方法为：

- 确定在估计任何范围查询的行数时可允许的最大错误百分比 P。
- 如果谓词是 BETWEEN，那么分位数数目应近似为 100/P；如果谓词是任何其他类型的范围谓词(<、<=、>或>=)，那么分位数数目应近似为 50/P。

例如，25 个分位数导致的最大估计错误对于 BETWEEN 谓词应为 4%，而对于“>”谓词则应为 2%。通常，至少指定 10 个分位数。对于极端不均匀的数据才需要 50 个以上的分位数。如果只需要高频值统计信息，那么将 *num_quantiles* 设置为 0。如果将此参数设置为“1”，那么由于值的整个范围只适合分位数，因而不收集分位数统计信息。

收集特定列的分布统计信息

为了提高 RUNSTATS 和后续查询方案分析的效率，可以仅收集查询在 WHERE、GROUP BY 和类似子句中使用的列的分布统计信息。还可以收集关于列的组合的基数统计信息。优化器使用这种信息，在为引用组中列的查询估计选择性时检测列相关。

注意：

RUNSTATS 仅收集执行该命令的数据库分区上的表的统计信息。将此数据库分区的 RUNSTATS 结果推广到其他数据库分区。如果执行 RUNSTATS 的数据库分区不包含表的一部分，那么将请求发送到数据库分区组中持有表的该部分的第一个数据库分区。

下面我们举一些收集特定列的分布统计信息的例子。

下面的例子说明了使用 RUNSTATS 来收集包含数据分布信息的系统目录表统计信息的不同方法。

例 8-26 收集表和索引上的数据库统计信息，包含分布统计信息。

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION AND INDEXES ALL
```

例 8-27 收集表上的数据库统计信息以及索引上的详细统计信息，包含分布统计信息。

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION AND DETAILED INDEXES ALL
```

例 8-28 收集选定列中包含分布的数据库统计信息。

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION ON COLUMNS(acctno,
acctname)
```

例 8-29 只收集表上的数据库统计信息，包含 acctno 和 acctname 列上的基本列统计信息，以及 mgrno 和 admrdept 列上的分布统计信息。

```
RUNSTATS ON TABLE db2inst1.account ON COLUMNS (acctno, acctname)
  WITH DISTRIBUTION ON COLUMNS (mgrno, admrdept)
```

例 8-30 收集构成索引的所有列，以及两个非索引列中包含分布的数据库统计信息。

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION ON KEY
  COLUMNS AND COLUMNS (admrdept, location)
```

下面的例子说明了使用 RUNSTATS 来收集目录统计信息和指定 *num_freqvalues* 与 *num_quantiles* 的不同方法。

例 8-31 收集包含分布统计信息的数据库统计信息。

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION AND INDEXES ALL
```

在上面的例子中我们没有指定分位数和频率统计信息，它会使用默认值，我们可以查询数据库配置参数来查到该默认值，如下所示：

```
/home/db2inst$db2 get db cfg for sample | grep -i "NUM "
 保留的高频值的数目                       (NUM_FREQVALUES) = 10
 保留的分位点数目                          (NUM_QUANTILES) = 20
```

例 8-32 仅收集表上的数据库统计信息，其中使用指定的 *num_freqvalues*，并从数据库配置设置选择 *num_quantiles* 收集所有列上的分布统计信息。

```
 RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION DEFAULT
NUM_FREQVALUES 40
```

将 *num_freqvalues* 参数设置为 40，将 *num_quantiles* 参数设置为 20。

例 8-33 收集表上的数据库统计信息，包含列 acctno 和 acctname 上的分布统计信息。单独为 acctname 列设置分布统计信息的范围，而 acctno 列使用公共的默认值。并为两个索引 IDX1 和 IDX2 收集数据库统计信息。

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION ON COLUMNS
   (acctno, acctname NUM_FREQVALUES 50 NUM_QUANTILES 100) DEFAULT
   NUM_FREQVALUES 5 NUM_QUANTILES 10 AND INDEXES db2inst1.IDX1, db2inst1.IDX2
```

对于 acctname 列，*num_freqvalues* 是 50，*num_quantiles* 是 100。

对于 acctno 列，*num_freqvalues* 是 5，*num_quantiles* 是 10。

例 8-34 收集所有索引上的数据库统计信息，包含列 acctname 上的分布统计信息。未列出的列上的分布统计信息将被清除。

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION ON COLUMNS
   (acctname NUM_FREQVALUES 20 NUM_QUANTILES 40)    AND INDEXES ALL
```

对于 acctname 列，*num_freqvalues* 是 20，*num_quantiles* 是 40。

例 8-35 收集所有索引以及列 acctno 和 acctname 上的数据库统计信息。acctno 列的 *num_freqvalues* 和 *num_quantiles* 的值将从默认值中获得。

```
RUNSTATS ON TABLE db2inst1.account WITH DISTRIBUTION ON COLUMNS
  (acctno, acctname NUM_FREQVALUES 20 NUM_QUANTILES 40) DEFAULT
   NUM_FREQVALUES 0 NUM_QUANTILES 0  AND INDEXES ALL
```

对于 acctname 列，*num_freqvalues* 是 20，*num_quantiles* 是 40。对于 acctno 列，*num_freqvalues* 是 0，*num_quantiles* 是 0。其他所有列不包含任何统计信息。

总的来说，分布统计信息对处理不均匀的数据分布非常有好处，它可以让优化器更好地选择最优的执行计划。在本书的最后一章有关于使用分布统计信息提高性能的案例，请读者仔细阅读该案例以增强对分布统计信息的理解。

8.1.7　统计信息更新策略

在下列情况下，使用 RUNSTATS 命令收集统计信息：

- 当向表装入数据并创建了新的索引时。
- 当用 REORG 命令重新组织表和索引时。
- 当存在大量影响表及其索引的更新、删除和插入操作时(此处的“大量”可能意味着 10%到 20%的表和索引数据都受到了影响)。
- 在绑定对性能要求很好的应用程序之前。
- 当您希望将新的和以前的统计信息进行比较时。定期进行统计使您能够在早期阶段发现性能问题。
- 当预存取大小(prefetchsize)发生变化时。
- 当在表中创建新的索引时。如果自从上次在表中运行 RUNSTATS 以来尚未修改表，那么只需要对新的索引执行 RUNSTATS。
- 使用 RUNSTATS 命令收集关于 XML 列的统计信息。如果仅使用 RUNSTATS 收集 XML 列的统计信息，将保留 LOAD 或上一次执行 RUNSTATS 命令已收集的非 XML 列的现有统计信息。如果先前已收集关于一些 XML 列的统计信息，那么在当前命令未收集关于这些 XML 列的统计信息时，将删除先前收集的 XML 列的统计信息；在当前命令收集了关于这些 XML 列的统计信息时，将替换先前收集的 XML 列的统计信息。

要提高 RUNSTATS 性能并节省用来存储统计信息的磁盘空间，可以考虑仅指定需要收集其数据分布统计信息的列。

如果您没有足够的时间一次收集全部的统计信息，那么可以运行 RUNSTATS 来每次仅更新几个表、索引或统计信息视图的统计信息，并轮流完成该组对象。如果对选择性部分更新运行 RUNSTATS 期间由于表上的活动而产生了不一致性，那么在查询优化期间将发出警告消息(SQL0437W，原因码 6)。例如，如果执行 RUNSTATS 来收集表分布统计信息，并且在某个表活动后，再次执行 RUNSTATS 来收集该表的索引统计信息，那么可能发生这

种情况。如果由于表上的活动产生了不一致，并且在查询优化期间检测到这些不一致，那么发出该警告消息。当发生这种情况时，应再次运行 RUNSTATS 来更新分布统计信息。

要确保索引统计信息和表同步，执行 RUNSTATS 来同时收集表和索引统计信息。索引统计信息保留自上次运行 RUNSTATS 以来收集的大部分表和列的统计信息。如果自上次收集该表的统计信息以来已对该表做了大量修改，那么只收集该表的索引统计信息将使两组统计信息不能在所有节点上都同步。

在生产系统中调用 RUNSTATS 可能会对生产工作负载的性能产生负面影响。RUNSTATS 命令现在支持优先级选项，在执行较高级别的数据库活动期间，可以使用优先级选项来限制执行 RUNSTATS 对性能的影响。

收集视图的统计信息时，将收集所有包含该视图引用的基本表的数据库统计信息。

可以考虑采用以下技巧来提高 RUNSTATS 的效率和已收集的统计信息的准确性：

- 仅对用来连接表的列或 WHERE、GROUP BY，以及查询的类似子句中的列收集统计信息。如果对这些列建立了索引，那么可以用 RUNSTATS 命令的 ONLY ON KEY COLUMNS 子句指定列。
- 为特定表和表中特定列定制 *num_freqvalues* 和 *num_quantiles* 的值。*num_freqvalues* 提供了重复最多的列和数据值的信息。*num_quantiles* 提供了数据值对于其他值而言是如何分布的有关信息。
- 使用 SAMPLED DETAILED 子句通过抽样计算详细的索引统计信息，这样就可以减少为获得详细索引统计信息而执行的后台计算量，使用 SAMPLED DETAILED 子句可以减少收集统计信息所需要的时间，并在大多数情况下产生足够的精度。
- 当创建已装载数据的表的索引时，添加 COLLECT STATISTICS 子句来在创建索引时创建统计信息，这一技巧在 Oracle 环境下也同样适用。
- 当添加或删除了大量数据时，或更新了收集其统计信息的列中的数据时，需要再次执行 RUNSTATS 命令来更新统计信息。
- 在 DB2 V9.5 之前，因为 RUNSTATS 仅收集单个数据库分区的统计信息，所以如果数据不是在所有数据库分区中一致分发的，那么统计信息将不太准确。如果您怀疑存在不一致的数据分发，那么您可能想要在执行 RUNSTATS 之前使用 REDISTRIBUTE DATABASE PARTITION GROUP 命令来在各数据库分区之间重新分发数据。

您可以通过比较查询 RUNSTATS 之前和之后的 SQL 语句的 EXPLAIN 输出，确定运

行 RUNSTATS 对访问计划的影响。

完成每一条 RUNSTATS 语句之后，您都应该执行显式的 COMMIT 命令。COMMIT 将释放锁，并避免在收集多个表的统计信息时填写日志。

在用 RUNSTATS 收集了统计信息之后，要使用 BIND 命令或 REBIND 命令重新绑定包含了静态 SQL 的应用程序包(并可以选择重新解释其语句)。db2rbind 命令可用于重新绑定数据库中的所有应用程序包。使用 FLUSH PACKAGE 命令来删除程序包缓存器(package cache)中当前所有缓存的动态 SQL 语句(db2 flush package cache dynamic)，并强制隐式地编译下一请求。

8.2　自动统计信息更新

DB2 优化器使用目录统计信息来确定查询的最佳访问方案。过期或者不完整的表或索引统计信息可能会导致优化器选择并非最佳的方案，从而导致查询执行速度下降。但是，决定要为给定的工作负载收集哪些统计信息是很复杂的事情，使这些统计信息保持最新也是一项很花费时间的任务。DB2 提供了自动收集统计信息功能(这是 DB2 的自动表维护功能的组成部分)，DB2 会自动监视表和定期收集活动量较大且导致统计信息更改的表的统计信息，后台进程以固定的时间间隔来评估表的活动，决定是不是需要更新统计信息。通过使用自动收集统计信息功能，可以让数据库管理器确定是否需要更新统计信息。DB2 还提供了实时收集统计信息功能，它的一个优点是当 SQL 提交到编译器时，优化器决定表的统计信息是否准确。如果不准确，将再次收集统计信息。也就是说，当需要表统计信息来优化和运行查询时，将自动收集表统计信息。通过新的动态配置参数 auto_stmt_stats 来启用实时自动收集统计信息功能。

8.2.1　自动 RUNSTATS 的基本概念

1. 自动 RUNSTATS 的自动收集统计信息功能

在启用自动收集统计信息功能之后，每隔两个小时执行一次异步收集统计信息检查操作。判断是否需要收集表统计信息的流程图如图 8-1 所示。

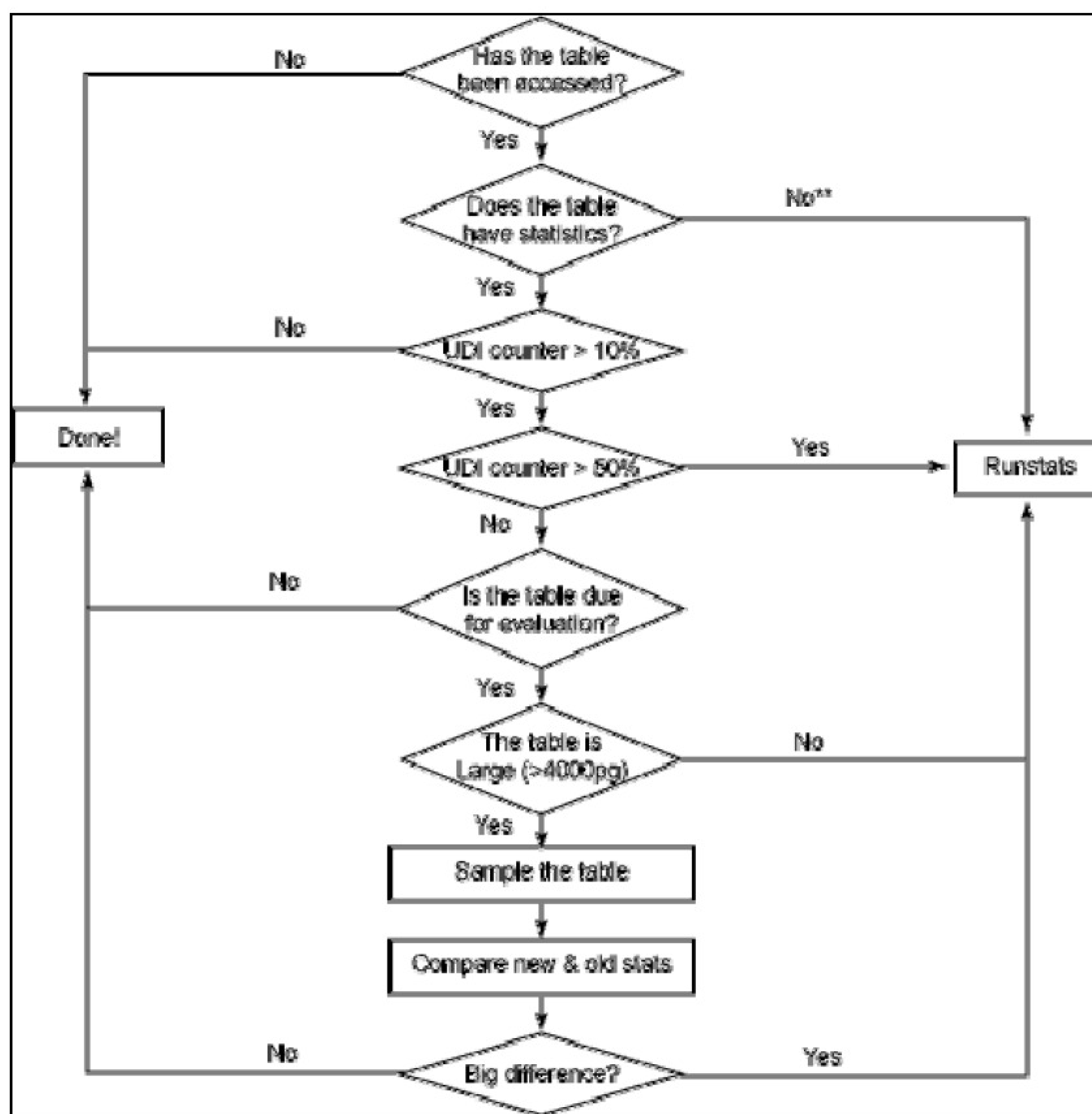


图 8-1 判断是否需要收集表统计信息的流程

注意：

我们不能修改 UDI(UPDATE、DELETE、INSERT)的比例和两小时间隔。在 RUNSTATS 执行过程中，无法中断 RUNSTATS 任务，除非停止实例或停用数据库。为了避免这种情况发生，我们最好在数据库维护窗口内完成 RUNSTATS。不管 util_impact_lim(影响策略)设置成多大，自动 RUNSTATS 始终使用 7%的固定值。

2. auto_stmt_stats 自动语句统计信息

auto_stmt_stats 参数启用和禁用收集实时统计信息，它是 auto_runstats 配置参数的子参数。仅当父 auto_runstats 配置参数也启用时，才启用此功能。在 OLTP 或迁移环境中，建议将该参数关闭。

```
  Automatic maintenance                     (AUTO_MAINT) = ON
    Automatic database backup             (AUTO_DB_BACKUP) = OFF
    Automatic table maintenance           (AUTO_TBL_MAINT) = ON
      Automatic runstats                 (AUTO_RUNSTATS) = ON
        Automatic statement statistics   (AUTO_STMT_STATS) = OFF
      Automatic statistics profiling     (AUTO_STATS_PROF) = OFF
        Automatic profile updates         (AUTO_PROF_UPD) = OFF
      Automatic reorganization              (AUTO_REORG) = OFF
```

3. health_mon 运行状态监视器

health_mon 参数是用来根据各种运行状况指示器来收集所选对象的运行状况的信息，而真正控制自动 RUNSTATS 的是数据库参数 auto_runstats。所以，当 health_mon 关闭而 auto_runstats 打开的时候，自动 RUNSTATS 仍然是可以运行的。反过来，health_mon 打开而 auto_runstats 关闭，在这种特殊的情况下，自动 RUNSTATS 会处于仅报告(reporting-only)的模式。不管自动 RUNSTATS 是否收集表的统计信息，监视器仍然会由于表的大量变化而显示该表需要重新收集统计信息。

4. 自动维护和状态监视器的关系

简而言之，数据库配置控制收集信息而状态监视器只控制报告。有趣的是，当 health_mon 关闭而 auto_runstats 打开的时候，自动 RUNSTATS 会收集需要统计的表信息，但是却无法通过状态监视器报告收集情况。不过，所有的收集统计活动都会记录在目录 $HOME/sqllib/db2dump/events 下面。

5. 默认的 RUNSTATS 命令

```
  runstats on table <inst>.<table> with distribution and sampled detailed
indexes all
```

另外，还可以为 RUNSTATS 建立统计信息的配置文件。统计信息配置文件是指一组选项，它预先定义了特定表上将要收集的统计信息。当将命令参数“SET PROFILE”添加到 RUNSTATS 命令时，将在表描述符和系统目录中注册或存储统计信息配置文件。

8.2.2　如何打开 auto_runstats

1. 打开自动维护开关

为了设置数据库自动进行统计信息收集，需要为自动维护开关设置数据库配置参数。其中，auto_runstats、auto_tbl_maint 和 auto_maint 必须打开，其他自动维护参数可选。

```
Automatic maintenance                          (auto maint) = on
  Automatic database backup                  (auto db backup) = off
  Automatic table maintenance                (auto tbl maint) = on
    Automatic runstats                       (auto runstats) = on
      Automatic statement statistics         (auto stmt stats) = off
    Automatic statistics profiling           (auto stats prof) = off
      Automatic profile updates              (auto prof upd) = off
  Automatic reorganization                   (auto_reorg) = off
```

2. 打开运行状态监视器(可选)

该参数是用来收集运行状况信息的，如果你关闭了该参数，就无法通过快照命令“db2 get health snapshot for database on <db_name>”查看收集到的信息。

```
db2 update dbmcfg using health_mon on
```

3. 创建自动维护策略文件

在目录<instance>/sqllib/tmp 下创建自动维护策略文件 DB2AutoRunstatsPolicy.xml，该文件可用来指定数据库管理器应如何执行自动表 RUNSTATS 操作。如果没有设置，默认会对包括系统表在内的所有表执行自动表 RUNSTATS 操作。

配置文件中需要填写的条件是来自 SQL 语句的过滤条件，如“select tabname from syscat.tables where <FilterCondition>”。

```
<?xml version="1.0" encoding="UTF-8"?>
<DB2AutoRunstatsPolicy
xmlns="http://www.ibm.com/xmlns/prod/db2/autonomic/config">
<RunstatsTableScope>
<FilterCondition></FilterCondition>
</RunstatsTableScope>
</DB2AutoRunstatsPolicy>
```

4. 创建自动维护窗口策略文件

在目录<instance>/sqllib/tmp 下创建 DB2MaintenanceWindowPolicy.xml，该文件用来指定数据库管理器应在其间安排自动维护的维护窗口，默认为每次。

```
<?xml version="1.0" encoding="UTF-8"?>
<DB2MaintenanceWindows
xmlns="http://www.ibm.com/xmlns/prod/db2/autonomic/config">
<OnlineWindow Occurrence="During" startTime="09:00:00" duration="8">
<DaysOfWeek>All</DaysOfWeek>
<DaysOfMonth>All</DaysOfMonth>
```

```
    <MonthsOfYear>All</MonthsOfYear>
    </OnlineWindow>
    </DB2MaintenanceWindows>
```

5. 应用自动维护策略

调用系统存储过程 automaint_set_policy 和 automaint_set_policyfile 来为数据库配置自动维护策略。

```
    db2 "call
sysproc.automaint set policyfile('auto runstats','DB2AutoRunstatsPolicy.xml')"
    db2 "call
sysproc.automaint set policyfile('MAINTENANCE WINDOW','DB2MaintenanceWindow
Policy.xml')"
```

6. 查看自动维护策略

调用系统存储过程 automaint_get_policy 和 automaint_get_policyfile 可将自动维护策略复制到新的文件中。

```
    db2 "call
sysproc.automaint get policyfile('auto runstats','DB2AutoRunstatsPolicy 2.xml')"
    db2 "call
sysproc.automaint get policyfile('MAINTENANCE WINDOW','DB2MaintenanceWindow
Policy_2.xml');"
```

7. 检查 RUNSTATS 运行状态监视器

必须打开 RUNSTATS 运行状态监视器才能看到 RUNSTATS 的评估报告。

```
    db2 get alert config for database on <db name>
```

8. 监控当前表的状态

```
    db2 get health snapshot for db on <db name>
    db2 get health snapshot for db on <db name> with full collection
```

8.2.3　如何监控 auto_runstats

为了解数据库发生了哪些统计信息收集活动，DB2 提供了统计信息日志。统计信息日志记录数据库的所有统计信息活动，包括自动和手动收集统计信息。统计信息日志的默认名称为 db2optstats.number.log，它位于$DIAGPATH/events 目录中。统计信息日志是旋转日志。统计信息日志行为由 db2_optstats_log 注册变量控制。

```
db2set db2_optstats_log=on,num=10,size=10
```

除了直接查看统计信息日志，还可以通过下面的 SQL 语句可以查看统计信息日志的内容：

```
select pid, tid,
substr(eventtype, 1, 10),
substr(objtype, 1, 30) as objtype,
substr(objname qualifier, 1, 20) as objschema,
substr(objname, 1, 10) as objname,
substr(first eventqualifier, 1, 26) as event1,
substr(second eventqualifiertype, 1, 2) as event2 type,
substr(second eventqualifier, 1, 20) as event2,
substr(third eventqualifiertype, 1, 6) as event3 type,
substr(third eventqualifier, 1, 15) as event3,
substr(eventstate, 1, 20) as eventstate
    from table(sysproc.pd get diag hist
      ('optstats', 'EX', 'NONE',
current timestamp - 1 year, cast(null as timestamp))) as sl
    order by timestamp(varchar(substr(first_eventqualifier, 1, 26), 26));
```

通过使用 db2pd -tcbstats，可以标识对表执行 UDI(UPDAT、DELETE 和 INSERT)操作的次数。

(a) New RTSUDI counter 是指从上一次收集实时统计信息、异步收集统计信息或手动收集统计信息后，增删改操作的次数。

(b) Existing UDI counter 是指从上一次异步收集统计信息或手动收集统计信息后，增删改操作的次数。

显示缓存中的统计信息：

```
db2pd -statisticscache summary | detail | find schema=<schema> object=<object>
```

8.2.4 自动收集统计视图的统计信息

在 DB2 V10 中，通过打开数据库参数 *auto_stats_views* 可以自动收集统计视图的统计信息。有时，为了提高查询性能，可考虑收集更高级的统计信息，例如列组统计信息或 LIKE 统计信息，或者创建统计视图。对统计视图的自动统计信息收集的启用或禁用通过使用 *auto_stats_views* 数据库配置参数完成。默认情况下，此数据库配置参数为 OFF，要启用此功能，请发出以下命令：

```
update dbcfg for dbname using auto_stats_views on
```

为自动收集统计视图的统计信息而发出的命令等价于以下命令：

```
runstats on view viewname with distribution
```

注意关键字是 view，并非 table，该参数是 DB2 V10 中新增加的。

8.3　碎片整理

随着数据的不断删除、插入和更新，数据页和索引页会变得越来越零散，数据页和索引页的物理存储顺序不再匹配其逻辑顺序，数据和索引结构的层次会变得过大，这些都会导致数据页跨越在多个页上和索引页的预读取变得效率低下。因此，需要根据数据更新的频繁程度适当地重新组织表和索引。

8.3.1　碎片产生机制和影响

对表数据进行大量更改之后，逻辑上连续的数据可能被分散存放在很多个不连续的物理数据页中，因此数据库管理器必须执行更多的读操作来访问数据。另外，对某个表删除大量行后，由于表的高水位标记并不会发生变化，因此对该表进行全表扫描时就会出现很多不必要的 I/O 操作。在这样的情况下，您可以考虑重组表以回收浪费的空间和对数据进行重组。此时，既可重组系统目录表，也可以重组数据库表。

提示：

由于重组表的时间通常要比运行统计信息的时间长，因此您可以执行 RUNSTATS 以更新当前的数据统计信息并重新绑定应用程序。如果更新的统计信息并没有改善性能，那么重组可能会有所帮助。

通过删除和插入操作对表进行更新后，索引的性能会降低，其表现形式如下：

- 索引叶子页分裂。叶子页被分裂之后，由于必须读取更多的叶子页才能访存表页，因此 I/O 操作成本会增加。
- 物理索引页的顺序不再与这些页上的键顺序相匹配(此称为不良集群索引)。叶子页出现不良集群情况后，顺序预取操作的效率将降低，因此会导致更多的 I/O 等待。
- 形成的索引大于其最有效的级别(level)数。在此情况下应重组索引。

如果在创建索引时设置了 MINPCTUSED 参数，那么在删除某个键且可用空间小于指定的百分比时，数据库服务器会自动合并索引叶子页。此过程称为联机索引整理碎片。但是，要复原索引集群和可用空间以及降低索引叶级别，请使用下列其中一种方法：

- 删除并重新创建索引。
- 使用 REORG INDEXES 命令联机重组索引。因为此方法允许用户在重建表索引期间对表进行读写操作，所以在生产环境中可能需要选择此方法。

- 使用允许脱机重组表及其索引的选项运行 REORG TABLE 命令。

8.3.2 确定何时重组表和索引

在对表数据进行许多更改之后，逻辑上连续的数据可能会位于不连续的物理数据页上，当许多更新操作创建了溢出(overflow)记录时更是如此。按这种方式组织数据时，数据库管理器必须执行额外的读操作才能访问顺序数据。另外，在删除大量数据后，空间并没有释放，也需要执行额外的读操作。

表重组操作会通过整理数据碎片来减少浪费的空间，并对行进行重新排序以合并溢出记录，从而加快数据访问速度并最终提高查询性能。还可以指定根据特定索引来重新排序数据，以便查询能通过最少的 I/O 读取操作就可以访问数据。

对表数据进行大量更改将导致更新索引并使索引性能下降。索引叶子页可能变成碎片或出现不良集群情况，并且索引有可能形成比所需层次(level)要多的层次以获得最佳性能(通常，几百万条记录的索引层次一般为 3，正常生产环境中索引的层次很少超过 4)。所有这些问题都会产生更多 I/O 并导致性能下降。

下列任何情况都需要我们重组表或索引：

- 自上次重组表之后，对查询访问的表进行了大量的插入、更新和删除活动。
- 对于使用具有高聚合度的索引的查询，其性能发生了明显变化。
- 在执行 RUNSTATS 以刷新统计信息后，性能没有得到改善。
- REORGCHK 命令指示需要重组表或索引(注意：在某些情况下，REORGCHK 始终建议重组表，即使在执行了重组后也是如此)。例如，如果使用 32KB 页大小，并且平均记录长度为 15 字节且每页最多包含 253 条记录，那么每页具有 32700－(15 × 253)=28905 个不可用字节。这意味着大约 88%的页面是可用空间。用户应分析 REORGCHK 的建议并针对执行重组所需的成本平衡利益。
- 综合考虑因查询性能不断降低浪费的成本和重组表所需的成本(包括 CPU 时间、重组的时间和 REORG 实用程序在完成重组操作之前锁定表所造成的并行性降低)，以确定是否进行表重组。

要确定是否需要重组表或索引，查询系统目录表中的统计信息并监视下列统计信息：

行的溢出(行链接)

查询 SYSSTAT.TABLES 视图中的 OVERFLOW 列以监视溢出值。当表中的可变长度列导致记录长度变长，以致它们不能放入数据页上的指定位置时，行数据就会溢出。在列添加至表定义并稍后通过更新行来更新该列时，长度也可能会更改。在这种情况下，行中的原始位置将保留一个指针，而实际值存储在由指针指示的另一个位置。这可能会影响性能，因为数据库管理器必须根据指针来查找行的内容。这个包括两个步骤的过程增加了处

理时间，并且还会增加需要的 I/O 数目。

重组表数据将消除行溢出；如果行链接数量较多，重组表数据的效果就会比较明显。

访存统计信息

当以索引顺序访问表时，查询 SYSCAT.INDEXES 和 SYSSTAT.INDEXES 系统目录表中的以下 3 个列来确定预取程序的效率。对照这些统计信息和基表来体现预取程序的平均性能的特征。

- AVERAGE_SEQUENCE_FETCH_PAGES 列存储可以按表中的顺序访问的平均页数。可以按顺序访问的页适合于预取。较小的数目指示预取程序没有充分发挥作用，原因是它们不能读入由表空间的预取大小设置指定的所有页数。较大的数指示预取程序将有效地执行。对于集群索引和表，此数目应该接近 NPAGES(包含一些行的页数)的值。
- AVERAGE_RANDOM_FETCH_PAGES 列存储当使用索引来访存表行时在顺序页访问之间的平均随机表页数。当大多数页按顺序排列时，预取程序忽略少量的随机页，并按已配置的预取大小继续预取。当表变得更加混乱时，随机访存页数就会增加。通常是由于在表的末尾或在溢出页中发生了无序的插入而导致这样的无组织。当使用索引来访问某一范围内的值时，这会导致降低查询性能的访存。
- AVERAGE_SEQUENCE_FETCH_GAP 列存储当使用索引进行访存时表页序列之间的平均间隔。通过扫描索引叶子页来进行检测，每个间隔表示在表页的各个序列之间必须随机访存的平均表页数。当随机访问许多页时发生这些情况，这会中断预取程序。较大的数指示无组织的表或较差集群的索引。

包含标记为已删除但未除去的 RID 的索引叶子页数

在 type-2 类型索引中，当 RID 标记为删除时，通常未在物理上删除 RID。这意味着可用空间可能会被这些逻辑上已删除的 RID 占用。要检索每个 RID 都被标记为已删除的叶子页的数目，查询 SYSCAT.INDEXES 和 SYSSTAT.INDEXES 统计信息表的 NUM_EMPTY_LEAFS 列。对于并非所有 RID 都标记为删除的叶子页，逻辑上已删除的 RID 的总数存储在 NUMRIDS_DELETED 列中。

使用此信息来通过执行带 CLEANUP ALL 选项的 REORG INDEXES 估计可以回收多少空间。要想只回收所有 RID 都被标记为已删除的页中的空间，执行带有 CLEANUP ONLY PAGES 选项的 REORG INDEXES。

索引的聚合比率和聚合因子统计信息

聚合比率统计信息存储在 SYSTCAT.INDEXES 系统目录表的 CLUSTERRATIO 列中。

此值(在 0 到 100 之间)表示使用索引的数据聚合的程度。如果收集 DETAILED 索引统计信息，0 和 1 之间的好的聚合因子统计信息存储在 CLUSTERFACTOR 列中，并且 CLUSTERRATIO 的值为–1，那么这两个集群统计信息中只有一个可以记录在 SYSCAT.INDEXES 目录表中。要将 CLUSTERFACTOR 值与 CLUSTERRATIO 值进行比较，可以将 CLUSTERFACTOR 乘以 100 以获得百分比。

注意：
通常，表中只有一个索引可以具有较高的聚合度。

如果查询用到的索引需要访问表中比较多的数据，而不是只访问索引或者只通过唯一索引访问表的一条记录，那么在具有较高聚合比率的情况下可能执行得更好。低的聚合度导致此类扫描要执行更多的 I/O，因为在每个数据页经过第一次访问后，下次访问该页时，该页仍在缓冲池中的可能性减小。增大缓冲池大小也可以提高非聚合索引的性能。

如果表数据最初是针对某个索引聚合的，而集群统计信息指示现在很少为同一索引聚合数据，那么您可能想重组该表以再次聚合数据。

索引叶子页数

查询 SYSCAT.INDEXES 表中的 NLEAF 列，以便了解索引占用的叶子页数目。该数目告诉您对索引进行完整扫描需要多少索引页 I/O。

理想情况下，索引应该尽可能占用最小的空间量，以便减少进行索引扫描所需要的 I/O 次数。随机的更新活动可导致页分割，这就增大了索引的大小。当在重组表期间重建索引时，可以使用最小空间量来构建每个索引。

注意：
默认情况下，当构建索引时，会在每个索引页上保留 10%的可用空间。要增加可用空间量，当创建索引时指定 PCTFREE 参数。无论何时重组索引，都使用 PCTFREE 值。

空数据页的数目

要计算表中的碎片数，查询 SYSCAT.TABLES 中的 FPAGES 和 NPAGES 列，并从 FPAGES 数减去 NPAGES 数。FPAGES 列存储正在使用的总页数；NPAGES 列存储包含一些行的页数。当删除整个范围内的行时，可能出现空页。

随着碎片的增多，就更需要进行表重组。重组表时将回收碎片并减少表使用的空间量。另外，因为会将碎片的数据页读入缓冲池以进行表扫描，所以回收未使用的页可以提高表扫描的性能。

8.3.3　执行表、索引检查是否需要做 REORG

reorgchk 命令返回有关数据组织的统计信息，并且可以在是否需要重组特定表或索引这一问题上为您提供建议。以下为 reorgchk 命令的输出：

```
    --------------------------------------------------------
    --DB2 CLP
    --------------------------------------------------------
    db2 reorgchk update statistics on table db2inst1.employee
    执行 RUNSTATS ....
    表统计信息：
    F1: 100 * OVERFLOW / CARD < 5
    F2: 100 * (Effective Space Utilization of Data Pages) > 70
    F3: 100 * (Required Pages / Total Pages) > 80
    SCHEMA    NAME    CARD    OV    NP    FP ACTBLK    TSIZE  F1  F2  F3 REORG
    -----------------------------------------------------------------------
    DB2INST1  EMPLOYEE          258500 51699 12932 16165
    - 61781500  19  93  80 *-*
    -----------------------------------------------------------------------
    索引统计信息：
    F4: CLUSTERRATIO 或正常化的 CLUSTERFACTOR > 80
    F5: 100 * (KEYS * (ISIZE + 9) + (CARD - KEYS) * 5) / ((NLEAF - NUM EMPTY
LEAFS) * INDEXPAGESIZE) > 50
    F6: (100 - PCTFREE) * ((INDEXPAGESIZE - 96) / (ISIZE + 12)) ** (NLEVELS -
2) * (INDEXPAGESIZE - 96) / (KEYS * (ISIZE + 9) + (CARD - KEYS) * 5) < 100
    F7: 100 * (NUMRIDS DELETED / (NUMRIDS DELETED + CARD)) < 20
    F8: 100 * (NUM EMPTY LEAFS / NLEAF) < 20
    SCHEMA   NAME                 CARD  LEAF ELEAF  LVLS
ISIZE  NDEL   KEYS  F4  F5  F6  F7  F8 REORG
    -----------------------------------------------------------------------
    表：DB2INST1.EMPLOYEE
    DB2INST1 IDX EMP C             258500 14894     0     4   106 21040
258500  72  48  13   7   0 **---
    SYSIBM   SQL060417152213950 258500  7122     0     4    60     0
258500  72  61  62   0   0 *----
```

在上面的输出中，我们重点关注“REORG”字段。如果 REORG 字段有一个或多个“*”，那么可以考虑对该表或索引重组。

使用 ORGANIZE BY 子句和相应的维索引定义的表的名称有'*'后缀。维索引的基数等价于表的“活动的块数”统计信息。

在重组之前，请综合考虑因查询性能不断降低浪费的成本和重组表或索引所需的成本(包括 CPU 时间、耗用时间和 REORG 命令在完成重组操作之前锁定表造成的并行性降低)，

以确定是否进行表重组。

DB2 提示信息说明

对 reorgchk 命令使用的度量的考虑因素包括(当查看 reorgchk 工具的输出时，找到用于表的 F1、F2 和 F3 这几列，以及用于索引的 F4、F5、F6、F7 和 F8 这几列。如果这些列中的任何一列有星号(*)，就说明当前的表和/或索引应该重组)：

F1：属于溢出记录的行所占的百分比。当这个百分比大于 5%时，在输出的 F1 列中将有一个星号(*)。

F2：数据页中使用了的空间所占的百分比。当这个百分比小于 70%(也就是说有 30%左右的碎片)时，在输出的 F2 列上将有一个星号(*)。

F3：其中含有包含某些记录的数据的页所占的百分比。当这个百分比小于 80%时，在输出的 F3 列上将有一个星号(*)。

F4：群集率，即表中与索引具有相同顺序的行所占的百分比。当这个百分比小于 80%时，在输出的 F4 列上将有一个星号(*)。指示索引需要 REORG，该索引与基本表不在相同的序列中。当在表中定义了多个索引时，一个或多个索引可能被标记为需要 REORG。指定 REORG 顺序的最重要索引。

F5：在每个索引页上用于索引键的空间所占的百分比。当这个百分比小于 50%时，在输出的 F5 列上将有一个星号(*)。

F6：可以存储在每个索引级的键的数目。当这个数字小于 100 时，在输出的 F6 列上将有一个星号(*)。

F7：在一个页中被标记为 deleted 的记录 ID(键)所占的百分比。当这个百分比大于 20%时，在输出的 F7 列上将有一个星号(*)。

F8：索引中空叶子页所占的百分比。当这个百分比大于 20%时，在输出的 F8 列上将有一个星号(*)。

提示：

在具体的生产实践中，reorgchk 命令的返回结果可以作为是否需要 REORG 的参考依据，但是并不能作为唯一的依据，具体是否需要做 REORG，还要根据表的实际业务行为做具体情况分析。

8.3.4 REORG 的用法和使用策略

REORG 命令的语法如下：

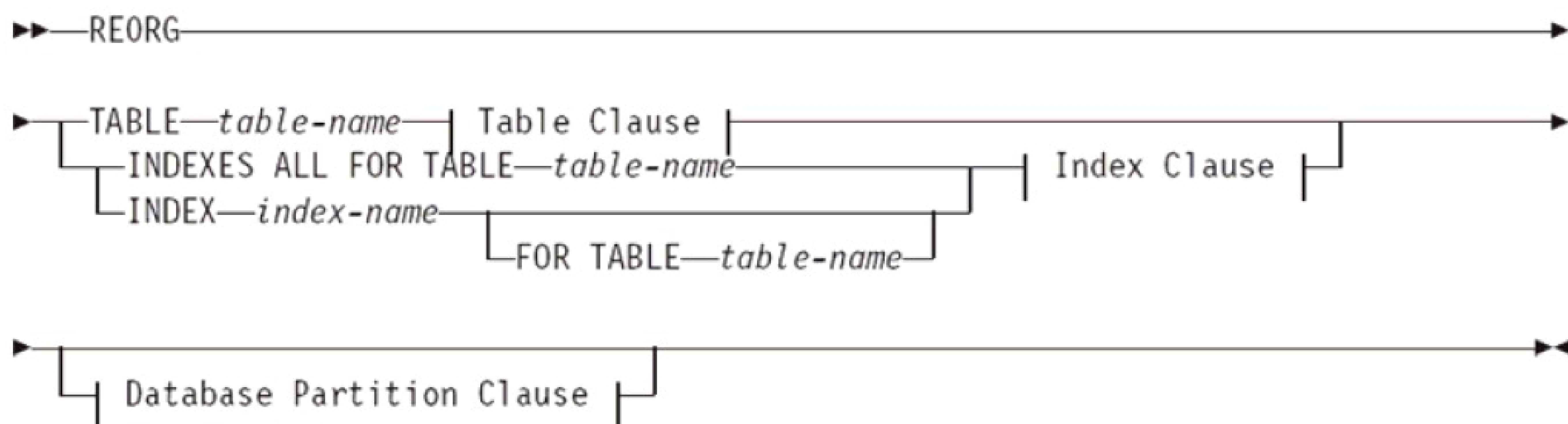
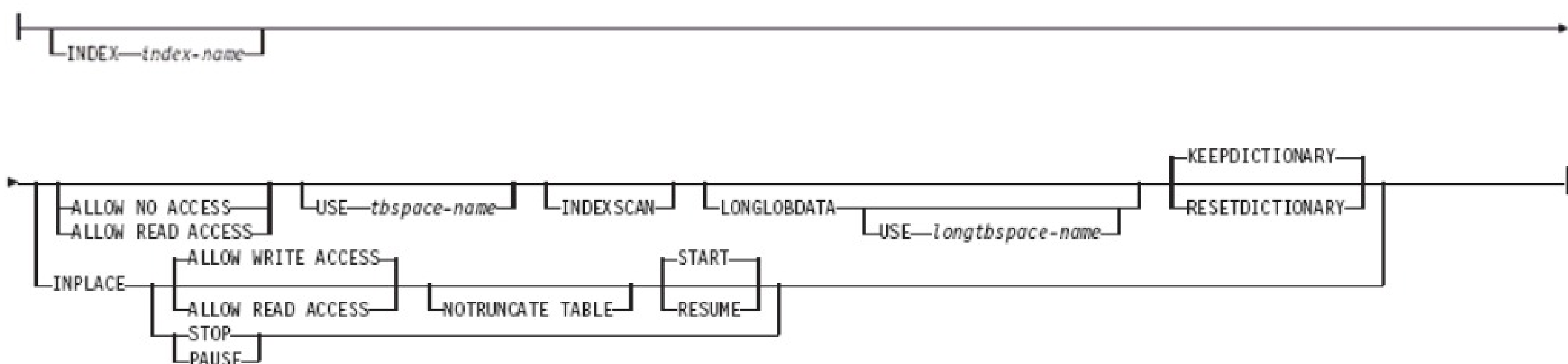
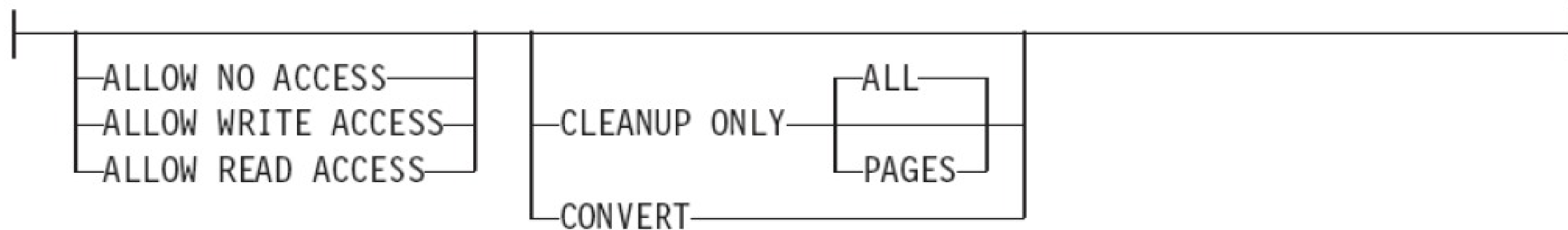


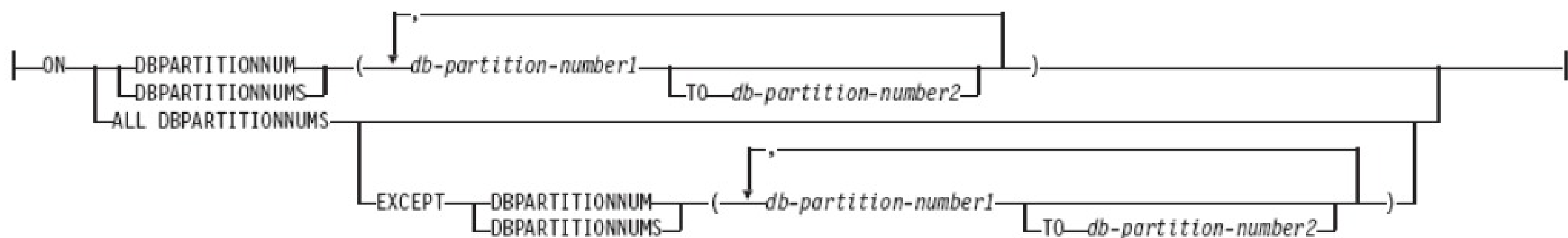
Table Clause:



Index Clause:



Database Partition Clause:



几个常用选项：

- USE tbspace-name：指定用来进行 REORG 的系统临时表空间，如果不指定的话，会使用当前表所在的表空间，表比较大的话有可能造成该表空间使用率达到 100%，影响其他表的访问
- RESETDICTIONARY：针对具有压缩属性的表进行压缩字典重建。
- KEEP DICTIONARY：针对压缩字典的表不进行压缩字典重建。
- INPLACE：指明需要使用联机重组。

DB2 提供了在线或离线执行 REORG 的选项。默认情况下，离线 REORG 允许其他用户读这个表，ALLOW READ ACCESS 是默认选项。可以通过指定 ALLOW NO ACCESS 选项来限制对表的访问。脱机表重组包括 4 个阶段，可以使用 db2pd -db dbname –reorgs 命令来查看。

1) 排序(scan-sort)：如果索引是使用 REORG TABLE 命令指定的，或者如果对表定义了集群索引，那么首先根据该索引对表行进行排序。如果指定了 INDEXSCAN 选项，那么使用索引扫描对表进行排序；否则，使用表扫描排序。

2) 构建(build)：在此阶段，将在要重组的表所在的表空间中或使用 REORG 命令指定的临时表空间中构建整个表的已重组副本。

3) 替换(replace)：在此阶段，通过从临时表空间中复制回表对象或通过在表所在的表空间中构建的对象来替换原始表对象。

4) 重新创建所有索引(Index Recreate)：重新创建在表上定义的所有索引。

在线 REORG (也称 inplace REORG)支持对表的读或写访问，ALLOW WRITE ACCESS 是默认选项。

表 8-2 是脱机重组和联机重组的比较结果。

表 8-2　脱机重组与联机重组的比较

特　　征	脱 机 重 组	联 机 重 组
性能	快	慢
完成时数据的集群因子	良好	非最佳集群
并行性(对表的访问)	ALLOW NO ACCESS ALLOW READ ACCESS(默认)	ALLOW READ ACCESS ALLOW WRITE ACCESS (默认)
数据存储空间要求	非常大	不是非常大
日志记录存储空间要求	不是非常大	非常大
用户控制(暂停和重新启动重组过程的能力)	较少控制	较多控制
可恢复性	完全可恢复或完全不可恢复：成功或失败	可恢复

(续表)

特　　征	脱 机 重 组	联 机 重 组
索引重建	进行	不进行
支持所有类型的表	是	否
指定除集群索引外的索引	是	否
使用临时表空间	是	否

DB2 V9.7 引入了新的特性，可以对分区表的单个分区进行表和索引重组，这个新特性使得用户可以仅对某个分区进行重组，并仅在该分区上对其他事务的读写权限进行限制。这样可以最大程度地缩小重组命令对其他事务的影响，提高事务的并发度。这样当用户使用分区重组时，在表上没有非分区索引的情况下，重组命令将完全只在一个分区上进行，从而对其他分区上的事务没有任何影响。相关的选项如下：

```
>>-REORG----------------------------------------------------->
--------------------------------------

>--+-------------------------------+------------------------->
   '-| Table partitioning clause |-'
-----------------------------------------------------
Table partitioning clause
|--ON DATA PARTITION--partition-name----------------------------|
```

对于生产数据库，如果有运维时间窗口，建议执行离线 REORG，并且在执行离线 REORG 时使用临时表空间(USE *tbspace-name*)。对于没有运维时间窗口的，可以尝试做在线 REORG，但是因为在线 REORG 的时间会特别长，所以需要人为控制，避开业务高峰期。

8.4　重新绑定程序包

重新绑定(rebind)是为先前绑定的应用程序重新创建程序包的过程。每个在数据库中执行的应用程序在执行之前都需要有一个绑定过程，这个绑定过程会根据数据库中各种统计信息和相关数据库对象的情况创建一个程序包，这个程序包中通常就是执行计划。所以在统计信息或相关数据库对象被修改之后，就需要对数据库中受影响的应用程序执行重新绑定，这样才能允许应用程序使用最新的更新。

那么具体哪些程序包必须执行重新绑定呢？DB2 的系统表 SYSCAT.PACKAGES 的 VALID 列标识当前的程序包是否可用，如果此列的值为“X”，就表示当前的程序包是不

可用的，此程序包就需要被重新绑定。这些程序包可以使用下面的方法进行绑定，也可以让数据库管理器在执行这样的程序包的时候自动地完成重新绑定(数据库管理器默认在隐式执行这些不可用的程序包的时候重新将它们绑定)。

建议在对统计信息更新之后对数据库中的应用程序执行重新绑定，以更新执行计划。还有一些其他情况需要及时地重新绑定程序包，比如创建了新的索引等。统计信息更新、碎片整理和 rebind 的顺序如图 8-2 所示。

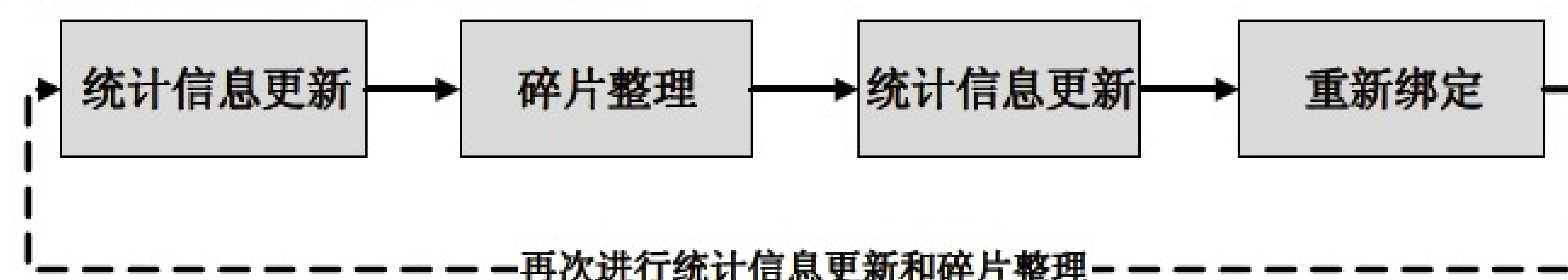


图 8-2　统计信息更新和碎片整理的处理流程

有两个命令可以执行重新绑定：

- rebind
- db2rbind

rebind 命令

rebind 命令可以重新创建数据库中已经存在的程序包。具体的语法如下：

```
>>-rebind--+---------+--package-name--------------------------->
           '-PACKAGE-'
>--+----------------------+--RESOLVE--+-ANY----------+--------->
   '-VERSION--version-name-'          '-CONSERVATIVE-'
>--+--------------+-------------------------------------------><
   +-REOPT NONE---+
   +-REOPT ONCE---+
   '-REOPT ALWAYS-'
```

db2rbind 命令

db2rbind 命令可以重新绑定数据库中存在的所有的程序包。具体语法如下：

```
>>-db2rbind--database-- -l--logfile--+-----+------------------->
                                      '-all-'
>--+----------------------------+--+----------------------+---><
   '- -u--userid-- -p--password-'  |      .-conservative-. |
                                   '- -r--+-any----------+-'
```

此命令的功能类似于 rebind，只是更简单一些，而且可以批量地处理数据库中所有的程序包。

8.5　本章小结

当数据库运行了较长时间之后，随着数据的变化或增长，数据库中的应用会变得越来越慢，我们就不得不对数据库进行统计信息更新、碎片整理和 rebind 以提高应用的性能。而且通常会一起考虑执行，并且应该遵循图 8-3 所示的处理流程。

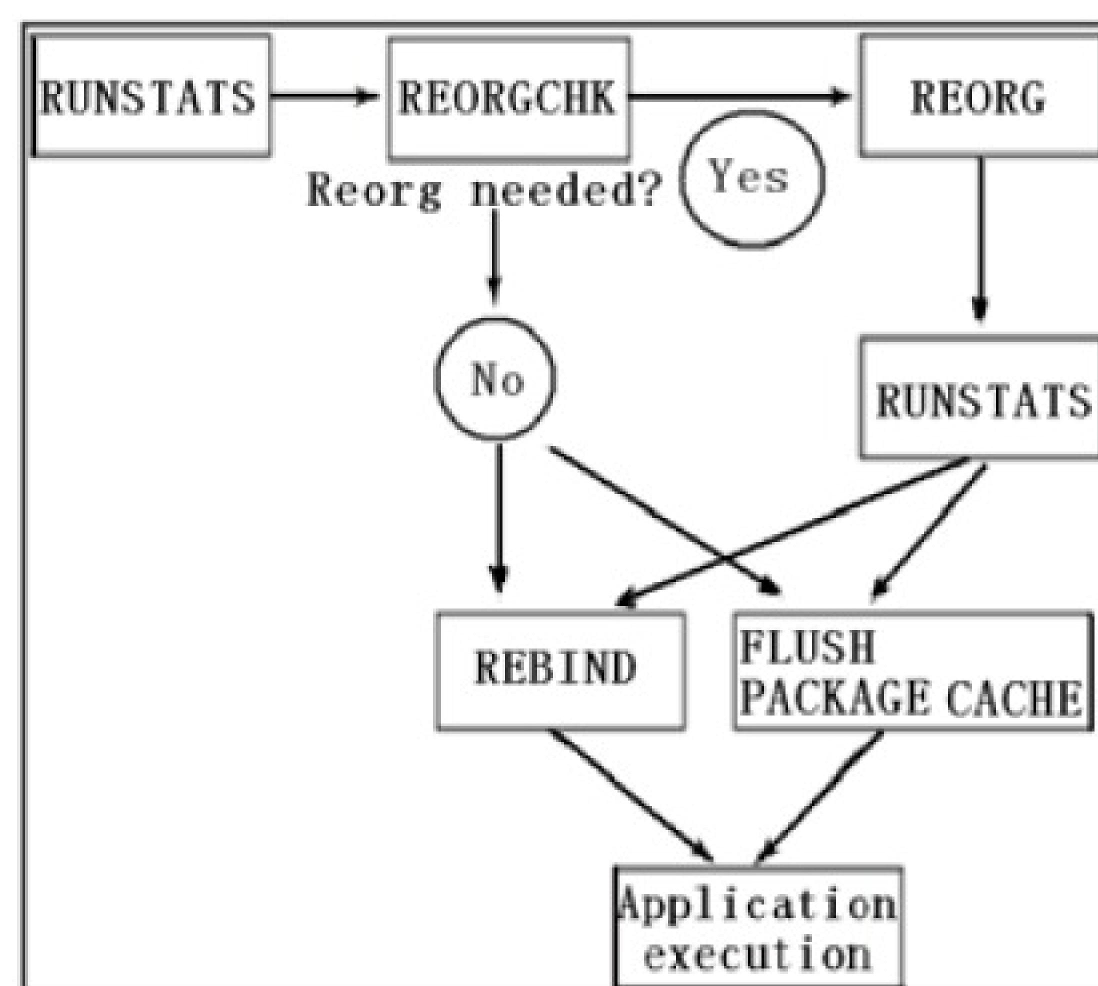


图 8-3　统计信息更新和碎片整理的处理流程

首先使用用于统计信息更新的 RUNSTATS 工具更新相关数据库对象的统计信息，然后使用数据库的 REORG 重组工具对数据库的碎片进行处理，接着再使用 RUNSTATS 工具对数据库的统计信息进行更新，最后将数据库中相关的程序包重新绑定。这样就完成了一个完整的统计更新和碎片整理的过程。在经过一定的时间周期后(一周、一个月或其他时间)，再次完成上面描述的完整过程。这样在数据库中访问数据的效率会得到及时更新。

那么上述循环过程的执行频率应该是何种频率呢？答案是：不一定。这需要根据实际情况来确定。最重要的就是我们要明确上述过程的最终目标是什么？上述过程的最终目标就是将数据库中数据的变化反映出来并让数据库做出相应的调整。所以数据库中数据变化得越快，上述过程的频率就应该越高；相反，数据库中数据变化得越慢，上述过程的频率也就越低。如果数据库中的数据没有变化，那么我们甚至不需要执行上述过程。还有个需要关注的因素就是时间窗口，实际的应用程序是否能够为我们执行上述过程留出足够的时间窗口。比如，对于 5×24(5 天×24 小时)的应用，那么平时是没有时间窗口让我们来完成上述过程的。这些工作只能放到周末去执行。所以最终采用何种频率来完成上面的过程，是由上面的两方面因素共同决定的。

第 9 章 SQL 语句调优

对于数据库管理员而言，调优数据库由于可能涉及各方面的问题，上到应用软件，下到操作系统、存储等，有时候还会涉及网络，因此这是一项复杂而极具挑战的工作。尽管复杂，但是大多情况下瓶颈都在一个点上，调优应用程序就是一种有效的方法，但在大多数生产系统中，DBA 很少甚至不能更改应用程序源代码，因此限制了他们调优应用程序的能力。通常最有效的调优方法是解决问题的根源，即从 SQL 语句本身入手。通常通过查找哪些 SQL 语句消耗的资源最多来获得有问题的 SQL，然后决定采取一定的 SQL 调优措施来减少资源消耗。

SQL 是标准的数据库查询语言。从某种意义上说，SQL 很简单，因为每个接触过数据库的人都会使用；但从另一个角度来看，SQL 也很复杂，因为在复杂的系统中，要想编写出简洁、高效的 SQL 语句并不是一件很容易的事。

在实际的生产运行中，我们发现往往引起数据库性能瓶颈的是那些比较耗时、耗资源的 SQL 语句，SQL 语句性能调优是数据库性能调整的重要任务之一。本章我们主要讲解 SQL 语句性能调优，主要内容包括：

- 通过监控定位引起性能资源的 SQL 语句
- 通过解释工具分析 SQL 语句执行计划
- 理解 SQL 工作方式
- SQL 调优案例
- 提高应用程序性能
- 高性能 SQL 语句注意事项

9.1 通过监控找出最消耗资源的 SQL 语句

在 DB2 数据库中，SQL 语句分为动态 SQL 语句和静态 SQL 语句，不管是哪一种，我们都可以通过查找 DB2 表函数 MON_GET_PKG_CACHE_STMT 来找出最耗时间的 SQL 语句和最耗 CPU 的 SQL 语句(具体语法和指标请参照 DB2 V10.5 知识中心)。

最耗时间的动态 SQL 语句：

```
    db2 "SELECT STMT_EXEC_TIME/NUM_EXEC_WITH_METRICS as AVG_EXEC_TIME,
STMT_TEXT FROM TABLE(MON_GET_PKG_CACHE_STMT('D', NULL, NULL, -2))as T WHERE
T.NUM_EXEC_WITH_METRICS <> 0 ORDER BY AVG_EXEC_TIME DESC"
```

最耗 CPU 的动态 SQL 语句：

```
    db2 "SELECT TOTAL_CPU_TIME/NUM_EXEC_WITH_METRICS as AVG_CPU_TIME,
STMT_TEXT FROM TABLE(MON_GET_PKG_CACHE_STMT ('D', NULL, NULL, -2)) as T WHERE
T.NUM_EXEC_WITH_METRICS <> 0 ORDER BY AVG_CPU_TIME DESC"
```

那么当我们找出这些耗时的 SQL 后，如何进行调整呢？下面我们将讲解如何调优 SQL。

9.2 通过解释工具分析 SQL 语句执行计划

将一条 SQL 语句提交给 DB2 数据库引擎进行处理时，DB2 优化器会对其加以分析，生成访问计划。访问计划包括将用于执行该语句的策略的详细信息(例如是否使用索引；若有排序方法，需要怎样的排序方法、连接方式、扫描方式等)。如果该 SQL 语句是嵌入在应用程序中编写的，那么在访问计划生成于预编译时，另外还会生成可执行形式的访问计划，作为称为“包”(package)的对象存储在系统编目表(SYSCAT.STATEMENTS)中。但如果语句是通过 CLP 提交的，或者语句是应用程序中的一条动态 SQL 语句(也就是说，这是一条在应用程序运行时构造的 SQL 语句)，那么访问计划将在该语句发出时生成，而生成的可执行代码则临时地存储在内存中(位于应用程序包缓冲区中(pckcache_sz))，而不是系统系统编目表。这样，如果发出了一条 SQL 语句，而程序包缓冲区中已有其可执行形式的访问计划，那么已有访问计划将被重用，不会再次调用 DB2 优化器，这提高了性能。

为什么说这非常重要？原因在于，尽管可以使用数据库系统监控器和活动监视器来获取关于某些 SQL 操作执行的情况有多好(或多糟)的信息，但不能用这些监控器来分析单独

SQL 语句。要执行此类分析，您必须能够捕获并查看存储于 SQL 语句的访问计划中的信息。而为了捕获并查看访问计划信息，我们必须使用 DB2 解释工具。

使用解释工具，您可以捕获并查看为特定 SQL 语句选择的访问计划的具体信息，还有可用于帮助确定编写不良的语句或数据库中弱点的性能信息。特别地，解释数据将帮助我们了解 DB2 优化器如何为满足查询而访问表和索引。解释数据还可用于评估采取的任何性能调优行动。实际上，只要您更改了数据库的某些方面、SQL 语句或与语句交互的数据库配置参数，都应收集并检查解释数据，弄清楚您的更改对性能产生了怎样的效果(如果有效果的话)。同时，理解一条 SQL 语句的执行计划可以帮助我们：

- 理解为查询选择的执行计划。查看用于优化 SQL 语句的数据库统计信息。
- 确定是否使用索引来访问表数据。
- 允许您进行“前”“后”对比，从而查看性能调优的效果。
- 获得访问计划执行的各操作的详细信息，包括各操作的预计成本。
- 辅助设计应用程序。
- 确定应何时重新绑定(rebind)应用程序。
- 辅助数据库设计。

9.2.1 解释表

首先创建一组特殊的表(解释表)，之后才能捕获解释信息。表 9-1 列出了所有解释表以及各表用于存储的信息。

表 9-1 解 释 表

表 名	内 容
EXPLAIN_ARGUMENT	包含各独立操作符的特征(如果存在的话)
EXPLAIN_INSTANCE	包含 SQL 语句的源的基本信息，还有关于环境的信息(EXPLAIN_INSTANCE 表是所有解释信息的主要控制表。其他解释表中的各行数据显式地链接到该表中的各行)
EXPLAIN_OBJECT	包含 SQL 语句生成访问计划所需的数据对象的信息
EXPLAIN_OPERATOR	包含 SQL 编译器为满足 SQL 语句所需的所有操作符
EXPLAIN_PREDICATE	包含确定特定操作符应用哪些谓词的相关信息
EXPLAIN_STATEMENT	包含在得到不同级别的解释信息时存在的 SQL 语句文本。用户输入的内容中包含原始 SQL 语句
EXPLAIN_STREAM	包含关于各单独操作符和数据对象之间存在的输入/输出数据流的信息(数据对象本身显示于 EXPLAIN_OBJECT 表中，而数据流中涉及的操作符可在 EXPLAIN_OPERATOR 表中找到)

在获得一条 SQL 语句的解释数据之前，请查看 DB2 安装目录下的'$DB2HOME/sqllib/misc/explain.ddl'(UNIX 或 Linux)或'%DB2HOME\sqllib\misc\explain.ddl'(WINDOWS)，以找到解释表的 DDL。然后必须使用跟调用 explain 工具的授权 ID 相同的模式定义一组解释表。执行：

```
db2 -tvf EXPLAIN.DDL
```

9.2.2 db2expln

当包含嵌入式 SQL 语句的源代码文件绑定到数据库时(无论是在预编译过程中还是在延迟绑定过程中)，DB2 优化器将分析遇到的每一条静态 SQL 语句，并生成相应的访问计划，访问计划随后以程序包的形式存储在数据库中(syscat.packages)。给定数据库名称、包名称、包创建者 ID、部分号(如果指定了部分号为 0，就处理包的所有部分)，db2expln 工具可为存储在数据库系统目录中的任何包解释并说明其访问计划。由于 db2expln 工具直接处理包而非全面解释数据或解释快照数据，因而通常用来获取那些已选定用于未捕获其解释数据的包的访问计划的相关信息。但由于 db2expln 工具仅可访问已存储在包中的信息，因而只能说明所选的最终访问计划的实现，不能提供特定 SQL 语句优化方式的信息。

如果使用额外的输入参数，那么 db2expln 工具还可用于解释动态 SQL(不包含参数标记的动态 SQL 语句)语句。

举个例子，对于数据库 SAMPLE 中的程序包 DB2INST1.P0203450，可使用下面的命令查看：

```
db2expln -d SAMPLE -g -c db2inst1 -p P0203450 -s 0 -t
```

其中：

-g 是指给出存取计划的图形输出(用字符模拟)。

-s 0 是指分析所有的 SQL 命令。

下面是相应的输出：

```
    IBM DB2 Universal Database SQL and XQUERY Explain Tool Processing package
DB2INST1.P0203450.
    ******************** PACKAGE *********************
    Package Name = DB2INST1.P0203450
    --------Prep Date = 2002/12/17
    --------Prep Time = 16:02:04
    --------Bind Timestamp = 2007-12-17-16.02.04.373971
    --------Isolation Level ---------= Cursor Stability
    --------Blocking--------------- = Block Unambiguous Cursors
    --------Query Optimization Class = 5
```

```
--------Partition Parallel -------= No
--------Intra-Partition Parallel = No
--------Function Path----------- = "SYSIBM", "SYSFUN", "DB2INST1"
Processing Section 1.
-------------------- SECTION ----------------------
-------------------- SECTION ----------------------
Section = 2
SQL Statement:
update STAFF set NAME = 'test'
where ID = 350
Estimated Cost = 75
Estimated Cardinality = 2
Access Table Name = DB2INST1.STAFF ID = 2,3
| #Columns = 2
| Relation Scan
| | Prefetch: Eligible
| Lock Intents
| | Table: Intent Exclusive
| | Row : Update
| Sargable Predicate(s)
| | #Predicates = 1
Update: Table Name = DB2INST1.STAFF ID = 2,3
End of section
优化器 Plan:
------UPDATE
------( -2)
------/-- \
-TBSCAN --Table:
--( 3) --DB2INST1
---| -----STAFF
Table:
DB2INST1
STAFF
......
```

可以用下面的命令找出数据库中存在的程序包：

```
    db2 "select pkgschema, pkgname from syscat.packages
where pkgname ='P0203450'"
```

上面例子中的程序包 DB2INST1.P0203450 是存储过程。

查看动态 SQL 的例子：

```
$db2expln -d sample -q "select * from employee" -t
```

```
IBM DB2 Universal Database SQL and XQUERY Explain Tool
******************** DYNAMIC ***************************************
==================== STATEMENT ==========================================
        Isolation Level          = Cursor Stability
        Blocking                 = Block Unambiguous Cursors
        Query Optimization Class = 5
        Partition Parallel       = No
        Intra-Partition Parallel = No
        SQL Path                 = "SYSIBM", "SYSFUN", "SYSP
                                   "ORACLE"
Statement:
  select *
  from employee
Section Code Page = 1208
Estimated Cost = 7.612985
Estimated Cardinality = 42.000000
Access Table Name = ORACLE.EMPLOYEE  ID = 2,6
|  #Columns = 14
|  Avoid Locking Committed Data
|  Relation Scan
|  |  Prefetch: Eligible
|  Lock Intents
|  |  Table: Intent Share
|  |  Row  : Next Key Share
|  Sargable Predicate(s)
|  |  Return Data to Application
|  |  |  #Columns = 14
Return Data Completion
End of section
```

9.2.3 db2exfmt

与 db2expln 工具不同，db2exfmt 工具设计用于直接处理已收集并存储在解释表中的全面解释数据或解释快照数据。给定数据库名和其他限定信息，db2exfmt 工具将在解释表中查询信息、格式化结果，并生成一份基于文本的报告，此报告可直接显示在终端上或写入 ASCII 文件。

所有 explain 输出都是从下往上读的，如图 9-1 所示。

这里将所有细节列在一个输出文件中。在图 9-1 中，每个操作符都编了号，当您往下查看时，每个操作符都将被详细解释。例如，图 9-1 中的一个操作符可以作如下解释：

读 Text Explain 操作符：

```
5.7904 – # of rows returned (based on statistics calculation)
HSJOIN  – type of operator
(   2) – operator #
75.536      – cumulative timerons
  3     – I/O costs
```

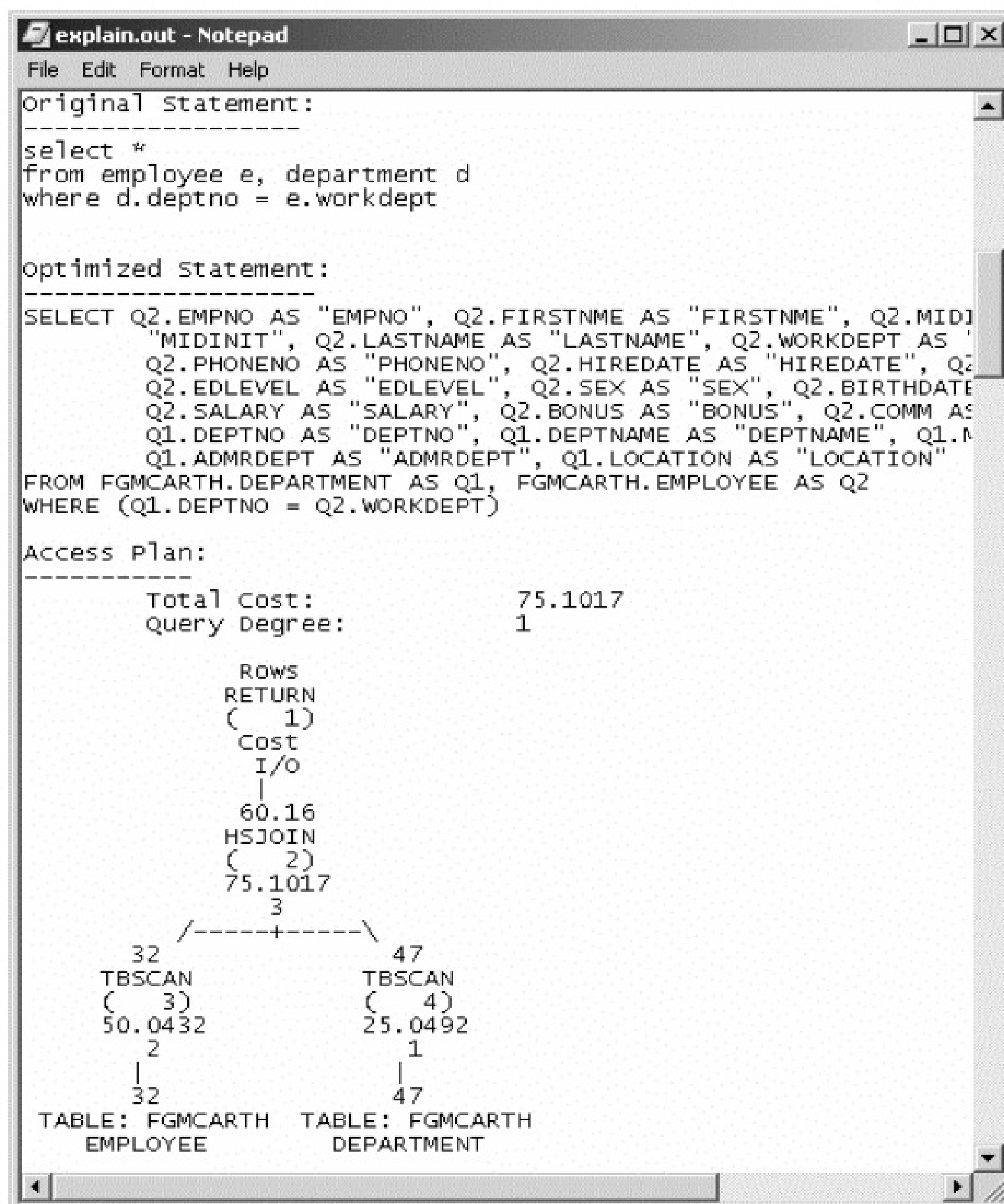


图 9-1　Text Explain 屏幕

返回的行数、timeron(cost)数和 I/O 都是优化器估计的，在某些情况下可能与实际数字不符。timeron 的概念我们前面已经讲过，它是 DB2 的成本度量单元，用于给出对数据库服务器在执行同一查询的两种计划时所需的资源或成本的粗略估计。估计时计算的资源包括处理器和 I/O 的加权成本。

您可以使用 db2exfmt 来解释单独一条语句。

为一条语句生成 Text Explain 输出：

```
explain all for
        SQL_statement
```

```
db2exfmt -d
        dbname -g tic -e
        explaintableschema -n % -s % -w -1 -# 0 -o
        outfile
```

如果为用“;”隔开的几条“explain all”语句构建文本文件，就可以一次解释多条语句。为多条语句生成 Text Explain 输出：

```
db2 -tf
        file with statements
db2exfmt -d
        dbname -g tic -e
        explaintableschema -n % -s % -w % -# 0 -o
        outfile
```

9.2.4 各种解释工具的比较

可用于显示全面解释数据和解释快照的不同工具有着很大的差异，无论是在复杂性方面还是在功能方面。表 9-2 对比了 db2expln 和 db2exfmt，并强调了各自的特征。要使解释工具发挥出最好的效果，应在选择工具时考虑实际的环境和需求。

表 9-2 可用解释工具的比较

所 需 特 征	db2exfmt	db2expln
用户界面	基于文本	基于文本
快速但粗略的静态 SQL 分析	否	是
静态 SQL 支持	是	是
动态 SQL 支持	是	是
CLI 应用程序支持	是	否
详细的 DB2 优化器信息可用	是	否
适于分析多条 SQL 语句	是	是

9.2.5 如何从解释信息中获取有价值的建议

当分析 explain 命令的输出信息时，应该注意以下问题：

- 对相同的一组列和基本表使用的 ORDER BY、GROUP BY 或 DISTINCT 操作符将从索引或物化查询表(MQT)中受益，因为有了索引，可以减少排序所消耗的时间。explain 可帮助确保索引被正确地用于连接谓词、本地谓词以及 GROUP BY 和 ORDER BY 子句，以尽量避免排序。

- 代价较高的操作，例如大型排序、排序溢出以及对表的大量使用，都可以受益于更多的排序空间(sortheap)、更好的索引、更新的统计信息或调优的 SQL。
- 部分表扫描也可以从索引中受益。
- 完全索引扫描或无选择性的索引扫描，其中不使用 start 和 stop 关键字，或者使用这两个关键字，但是有着很宽的取值范围。这样的扫描性能会很差。
- 表的连接类型。根据对表中数据的了解来确定连接类型是否正确，以及用于连接的内部表和外部表的表是否正确。
- 未充分地利用索引。查询是否按期望使用了索引？此问题可通过查看执行计划轻松应对。如果确实存在索引，就检查基数或索引键的顺序。确保索引的集群度。
- 表基数和'SELECT *'的使用。有时，根据要返回的列数，DB2 优化器会判定扫描整个表的速度更快。
- 优化级别的设置是否合适。

9.3　理解 SQL 语句如何工作

SQL 语句在执行过程中，谓词、分组排序、扫描方式和连接类型等都会对 SQL 语句的执行产生重大的影响，所以我们要想对 SQL 进行调优，就必须先了解这几个概念。

9.3.1　理解谓词类型

SQL 的另一个灵活特性是可以用不同的写法来完成相同的功能。例如，SQL 可以用关联或嵌套查询来达到相同的目的。始终可以将嵌套查询转换成等价的连接。可以在大量的函数和谓词中看到这一灵活性的其他示例。具有等价功能的特性的示例包括：

- BETWEEN 与<= / >=
- IN 与一系列和 OR 配合的谓词
- INNER JOIN 与 FROM 子句中串在一起并用逗号分隔的表
- OUTER JOIN 与带有 UNION 的简单 SELECT 和相关的子查询
- CASE 表达式与复杂的 UNION ALL 语句
- not in 与 not exists

SQL 展示的这一灵活性并不总是称心如意，因为写法不同但功能相同的 SQL 谓词可能具有不同的性能。

用户使用查询语句访问数据库，该查询语句需要用谓词限定符来进行数据的过滤。这

些限定符通常出现在查询的 WHERE 子句中。这种限定符称为谓词。可以将谓词分组为 4 种类别，按如何在求值过程中使用谓词以及何时使用来确定这些类别。这些类别列示如下，按最佳性能到最差性能的顺序排列：

- 范围定界谓词
- 索引控制谓词
- 数据控制谓词
- 保留谓词(SARGable predicate)

在数据库内部，谓词性能的层次如图 9-2 所示。

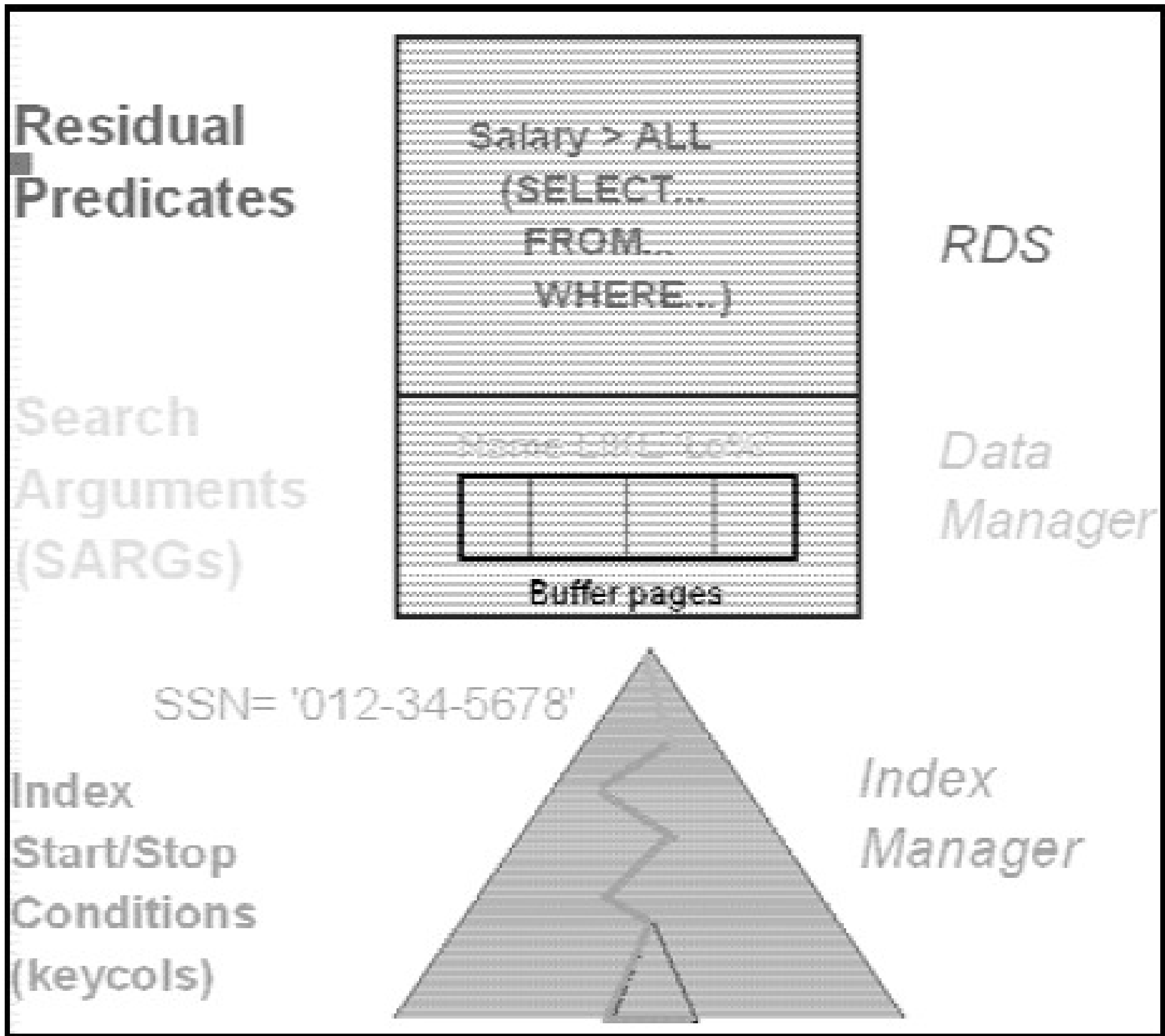


图 9-2　谓词性能的层次

看了上面的图 9-2，读者应该明白我们在写 SQL 语句时，应该尽量用最下面的谓词类别，因为越往上，谓词的性能就越差。

表 9-3 概述了谓词类别。后面会更详细地描述每个类别。

表 9-3　谓词类型特征的摘要

特　　征	谓 词 类 型			
	范围定界	索引控制	数据控制	保　　留
减少索引 I/O	是	否	否	
减少数据页 I/O	是	是	否	
减少内部传送的行数	是	是	是	
减少合格行的数目	是	是	是	

范围定界谓词和索引控制谓词

范围定界谓词限制索引扫描的范围。它们提供索引搜索的开始和停止键值。索引控制谓词不能限制搜索范围，但可根据该索引来求值，因为在该谓词中涉及的列是索引键的一部分。例如，考虑下列索引：

```
INDEX IX1:  NAME    ASC,
                              DEPT    ASC,
                              MGR     DESC,
                              SALARY  DESC,
                              YEARS   ASC
```

也考虑包含下列 WHERE 子句的查询：

```
WHERE  NAME   = :hv1
        AND  DEPT = :hv2       AND   YEARS > :hv5
```

前两个谓词(NAME = :hv1 和 DEPT = :hv2)是范围定界谓词，而 YEARS > :hv5 是索引控制谓词。

优化器在对这些谓词求值时读取索引数据而不是读取基本表。这些索引控制谓词减小需要从表中读取的行集，但它们不影响访问的索引页数。

XSCAN 数据运算符扫描也支持 XMLEXISTS 和 XMLTABLE 表达式中出现的 XML 数据上的谓词。索引范围扫描也支持这些谓词中的一部分谓词。

数据控制谓词

如果在索引页里取不到值，必须去数据页取值的谓词称为数据控制谓词。这些谓词通常需要访问表中的单独行。

例如，考虑在 PROJECT 表上定义的单个索引：

```
INDEX IX0:  PROJNO ASC
```

对于下列查询，因为需要过滤 DEPTNO = 'D11'的数据，但表 PROJECT 上只有字段 PROJNO 的索引，所以需要去数据页里查找 DEPTNO = 'D11'的数据，将 DEPTNO = 'D11'谓词视为数据 SARGable：

```
SELECT PROJNO, PROJNAME, RESPEMP
            FROM PROJECT
         WHERE DEPTNO = 'D11'
          ORDER BY PROJNO
```

保留谓词

保留谓词比访问表需要更多的 I/O 成本。它们可具有下列特征：

- 使用相关子查询
- 使用量化子查询，这些子查询包含 ANY、ALL、SOME 或 IN 子句
- 读取 LONG VARCHAR 或 LOB 数据，该数据存储在与表分开的文件中

这种谓词由“不走索引扫描，走全表扫描”来求值。

有时，当访问数据页时，必须重新应用仅应用于索引的谓词。例如，当访问数据页时，使用索引 OR 运算或索引 AND 运算的访问方案始终要将这些谓词作为保留谓词重新应用。

谓词总结：下面的图 9-3 给读者列出了谓词类型，哪些是可索引的谓词；哪些谓词即使存在索引，也是不会用到索引的。

Predicate Type 谓词类型	**Indexable**
INDEXABLE 可索引谓词	
COL = *value*	Y
COL = *noncol expr*	Y
COL IS NULL	Y
COL *op value*	Y
COL *op noncol expr*	Y
COL BETWEEN *value1* AND *value2*	Y
COL BETWEEN *noncol expr1* AND *noncol expr2*	Y
COL LIKE '*pattern*'	Y
COL IN (*list*)	Y
COL LIKE *host variable*	Y
T1.COL = T2.COL	Y
T1.COL *op* T2.COL	Y
COL=(*non subq*)	Y
COL *op* (*non subq*)	Y
COL *op* ANY (*non subq*)	Y
COL *op* ALL (*non subq*)	Y
COL IN (*non subq*)	Y
COL = *expression*	Y
(COL1,...COL*n*) IN (*non subq*)	Y
NON-INDEXABLE 不可索引谓词	
COL <> *value*	N
COL <> *noncol expr*	N
COL IS NOT NULL	N
COL NOT BETWEEN *value1* AND *value2*	N
COL NOT BETWEEN *noncol expr1* AND *noncol expr2*	N
COL NOT IN (*list*)	N
COL NOT LIKE ' *char*'	N
COL LIKE '%*char*'	N
COL LIKE '_*char*'	N
T1.COL <> T2.COL	N
T1.COL1 = T1.COL2	N
COL <> (*non subq*)	N

图 9-3　谓词类型

9.3.2　排序和分组

排序分为两个阶段：排序阶段和排序结果的返回阶段。

排序阶段

排序可以是溢出的，也可以是非溢出的。如果无法将排序的数据整个放入排序堆中(sortheap，排序堆是每次执行排序时分配的一块私有内存)，数据就会溢出到数据库所有的临时表中。不溢出的排序通常执行得比那些溢出的好。

排序结果的返回阶段

返回可以是管道的(piped)，也可以是非管道的(non-piped)。如果排序信息可以直接返回，而无需临时表来存储最终排序的数据列表，那它就是管道排序(piped sort)。如果排序信息的返回需要临时表，那它就是非管道排序(non-piped sort)。管道排序通常执行得比非管道排序好。

当优化器为 SQL 语句选择访问方案时，它会考虑对数据排序给性能带来的影响。当没有索引满足请求的对已访存的行的排序时，执行排序。当优化器确定排序比索引扫描开销较少时，也会进行排序。优化器采用下列其中一种方式来对数据进行排序：

- 当执行查询时，通过管道传送排序结果。
- 在数据库管理器内对排序进行内部处理。

排序时间取决于多种因素，包括要排序的行数、键大小和行宽。如果要排序的行占用的空间超过排序堆(sortheap)中可用的空间，那么执行几遍排序，每一遍都要对整个行集的某个子集排序。每遍排序的结果存储在缓冲池中的一个临时表内。如果缓冲池里没有足够的空间，可以将此临时表中的页写到磁盘。当所有排序都完成时，必须将这些已排序的子集合并为单个已排序的行集。

排序中的分组操作

当排序产生 GROUP BY 操作所需的顺序时，优化器在排序时可执行 GROUP BY 聚集的部分或全部。如果每组中的行数较大，这种做法就有利。如果在排序期间执行一些分组以减少或消除排序溢出到磁盘的情况，就会更加有利。

排序中的聚集需要以下 3 个聚集阶段以确保返回正确结果。

(1) 聚集的第 1 个阶段“部分聚集”计算聚集值，直到填满排序堆为止。在部分聚集中，接受未聚集数据并产生部分聚集。如果已填满排序堆，其余数据会溢出到磁盘，包括在当前排序堆中已计算的所有部分聚集。在复位排序堆之后，开始新的聚集。

(2) 聚集的第 2 个阶段“中间聚集”提取所有溢出的排序运行结果，并根据分组键进一步聚集。由于分组键列是分布键列的子集，因而不能完成聚集。中间聚集使用现有的部

分聚集来产生新的部分聚集。此阶段并不总是会发生。它用于分区内和分区间并行性。在分区内并行性情况下，当全局分组键可用时完成分组。在分区间并行性情况下，如果分组键是用来在数据库分区间分组的分布键的子集，从而要求重新分发来完成聚集时，会发生这种情况。当多个代理程序在缩减为单个代理程序以完成对其溢出的排序结果的合并时，在分区内并行性中将存在相似的情况。

(3) 聚集的最后阶段“最终聚集”使用所有部分聚集并产生最终聚集。此步骤始终通过 GROUP BY 运算符执行。排序不能执行完整的聚集，因为它们不能保证排序不会被分割。完整的聚集接受未聚集的数据，然后产生最终聚集。如果分发不能禁止它的使用，这种聚集方法通常用于将已处于正确顺序的数据进行分组。

排序和分组是比较消耗资源的，所以在 SQL 语句中尽量避免无谓的分组和排序。关于排序的监控和相关配置参数的配置请参见本书“第 4 章：DB2 配置参数调整”。

9.3.3 连接方法

如果查询包含不止一个表，那么就应该使用连接谓词来连接那些表以避免笛卡尔(Cartesian)连接。当评估查询执行计划时，优化器计算并比较每种连接方法的成本，然后选择要使用的最佳方法。最常用的连接方法就是嵌套循环连接、合并连接和哈希连接。

嵌套循环连接

在嵌套循环连接中，将扫描第一个(或外部)表以查找满足查询规则的行。对于在外部表中找到的每一行，数据库服务器将在第二个(或内部)表中搜索相应的行。通过索引扫描还是表扫描来访问外部表则取决于该表。如果有过滤条件，数据库服务器首先会应用它们。如果内部表没有索引，那么数据库服务器就会将在表上构建索引的成本与连续扫描的成本进行比较，然后选择成本最低的那一种方法。总成本取决于连接列上是否有索引。如果连接列上有一个索引，那么成本会相当低；否则，数据库服务器就必须对所有表(外部和内部表)执行表扫描。

合并连接

当连接表的连接列上没有可用索引时，通常使用该连接方法。连接开始之前，如果有过滤条件，那么数据库服务器首先会应用它们，然后对连接列上每个表中的行进行分类。一旦实现对行的分类，连接两个表的算法就十分容易：数据库服务器仅仅连续地读取两个已分类表，并合并所有相匹配的行。因为该方法在进行表连接之前，必须将所有的连接表分类，所以成本通常极高。

哈希连接

当一个或多个连接表上没有索引时，或者当数据库服务器必须从所有连接表中读取大

量行时，就使用这种方法。在该方法中，需要扫描其中的一个表，通常扫描较小的那个表，用它在内存中创建哈希表。通过哈希函数，将具有相同哈希值的行放在内存中。在扫描完第一个表并将它放在哈希表中之后，就扫描第二个表，并在哈希表中查找该表中的每一行，看是否可以进行连接。如果连接中有更多表，那么数据库服务器将对每个连接表执行相同的操作。

哈希连接包含两个动作：构建哈希(或者是我们所称的构建阶段)以及探测哈希表(或探测阶段)。在构建阶段，数据库服务器读取一个表，并且在应用所有现有过滤条件之后，在内存中创建哈希表。可以在概念上将哈希表认为是一系列的内存 bucket，每个 bucket 所拥有的地址是通过应用哈希函数从键值导出的。数据库服务器不会在特定的哈希 bucket 中对键进行分类。在探测阶段，数据库服务器将读取连接中的其他表，如果存在连接谓词，就应用它们。在满足连接谓词限定条件的每个行中，数据库服务器将对键应用哈希函数，并探测哈希表以查找匹配的键值。哈希连接通常比分类合并连接快，因为没有涉及分类操作。

9.3.4　扫描方式

扫描方式是指优化器从数据库表中读取(更确切地说是检索)数据的方法。基本上有两种方法：

- 最简单的方法是连续读取表中的数据，就是按照我们通常调用它的方法来执行表扫描。当无论如何都必须读取表中大多数数据时，或者当表没有索引时，优化器就会选择执行全表扫描。
- 另一种方法就是使用索引。如果一些列上有索引，那么优化器也许能使用索引扫描(index scan)。

表 9-4 对这两种方式进行了总结。

表 9-4　全表扫描与索引扫描

扫 描 方 式	描　　述
IXSCAN	使用可选的启动/停止条件扫描表索引，产生有序的行流
TBSCAN	通过直接从数据页中读取所有数据来检索行

优化器将比较每种方法的成本，确定使用最好的一种。它将评估查询执行成本的方方面面，例如所需的磁盘 I/O 操作数目、将被检索的行数、排序成本等。

通常，在返回大数据量的情况下，我们应尽量选用索引扫描，关于全表扫描和索引扫描的详细信息，请参考本书“第 7 章：DB2 优化器”。

9.4 SQL 调优案例

9.4.1 尽量使用单条语句完成逻辑

让我们从一个简单的编码技巧开始。如下所示的单个 INSERT 行序列：

```
INSERT INTO tab comp VALUES (item1, price1, qty1);
INSERT INTO tab comp VALUES (item2, price2, qty2);
INSERT INTO tab_comp VALUES (item3, price3, qty3);
```

可以改写成：

```
INSERT INTO tab comp VALUES (item1, price1, qty1),
                            (item2, price2, qty2),
                            (item3, price3, qty3);
```

执行这个多行 INSERT 语句所需时间大约是执行原来 3 条语句的三分之一。孤立地看，这一改进看起来似乎是微乎其微的。但是，如果这一代码段是重复执行的(例如该代码段位于循环体或触发器中)，那么改进是非常显著的。

类似地，如下所示的 SET 语句赋值序列：

```
SET A = expr1;
SET B = expr2;
SET C = expr3;
```

可以写成一条 VALUES 语句：

```
VALUES expr1, expr2, expr3 INTO A, B, C;
```

如果任何两条语句之间都没有相关性，那么这一转换保留了原始序列的语义。为了说明这一点，请考虑：

```
SET A = monthly avg * 12;
SET B = (A / 2) * correction_factor;
```

将上面两条语句转换成：

```
VALUES (monthly_avg * 12, (A / 2) * correction_factor) INTO A, B;
```

转换后的语句不会保留原始的语义，因为是以“并行”方式对 INTO 关键字之前的表达式进行求值的。这意味着赋给 B 的值并不以赋给 A 的值为基础，这是原始语句预期的语义。这样可以提高 SQL 语句性能。

9.4.2　合理使用 NOT IN 和 NOT EXISTS

一般情况下 NOT EXISTS 具有快于 NOT IN 的性能，但是这并不绝对。根据具体的数据情况、存在的索引以及查询的结构等因素，两者会有较大的性能差异，开发人员需要根据实际情况选择适当的方式。

例如下面的查询：

```
表结构：inst1.acct(acct id,name,balance) 主键：acct id
表结构：inst1.contact(contact id, acct id, point) 主键：contact id
表结构：inst1.contact detail(contact id, address, phone) 主键：contact id
查询 ：
select balance   from inst1.acct a
where not exists (select 1 from inst1.contact c
where a.acct id = c.acct id)
代价(cost)：139,109 timerons
```

此查询用来列出所有不存在联系人的账户余额。对于这样的需求，开发人员会很自然地写出上面的查询。对于大部分情况，上述查询具有最优的性能。让我们再来看看使用 NOT IN 后查询的总代价：

```
查询：
select balance   from inst1.acct a
where a.acct id not in (select c.acct id from inst1.contact c)
代价(cost)：8,021,703,168 timerons
```

可以看到 NOT EXISTS 的性能要比 NOT IN 高出许多。NOT IN 是自内向外的操作，即先得到子查询的结果，然后执行最外层的查询，而 NOT EXISTS 恰好相反，是自外向内的操作。在上述例子中，inst1.contact 表中有 100 万条记录，使得查询中的 NOT IN 列表非常大，导致使用 NOT IN 的查询具有非常高的查询代价。下面对上述查询进行修改，将 inst1.contact 表中的记录限制到 100 条，请看下面的查询：

```
查询：
select balance  from inst1.acct a
where not exists (select 1 from inst1.contact c
where  a.acct id = c.acct id
and c.contact id < 100)
代价：29,500 timerons
```

```
查询：
select balance   from inst1.acct a
where a.acct_id not in (select c.acct_id from inst1.contact c where
```

```
c.contact id < 100);
    代价：70,530 timerons
```

从上面可以看出：NOT EXISTS 的查询代价随子查询返回的结果集的变化没有大幅度下降，随着子查询的结果集从 100 万下降到 100 条，NOT EXISTS 的查询代价从 139109 下降到 29500，只下降 4 倍。但是 NOT IN 的查询代价却有着极大的变化，其查询代价从 8021703168 下降到 70530 下降了 113734 倍。可见子查询的结果集对 NOT IN 的性能影响很大，但是这个简单的查询不能说明 NOT EXISTS 永远好于 NOT IN，因为同样存在一些因素对 NOT EXISTS 的性能有很大的影响。我们再看下面的例子：

```
    查询：
    select balance   from inst1.acct a
    where not exists (select 1 from inst1.contact c
    where  a.acct id = c.acct id
    and  c.contact id in (select contact id from inst1.contact detail
    where contact id<100))
    代价：1,367,979 timerons
```

```
    查询：
    select balance   from inst1.acct a
    where a.acct id not in (select c.acct id from inst1.contact c
    where c.contact id in (select contact id from inst1.contact detail
    where contact id<100))
    代价：449,568 timerons
```

在上面的例子中，我们只是对查询增加了一处小的改动，使用嵌套查询限制了在 inst1.contact 中扫描的范围。但是在这两个新的查询中，NOT IN 的性能却又好于 NOT EXISTS。NOT EXISTS 的代价增加了 46 倍，而 NOT IN 的代价却只增加了 6 倍。这是由于 NOT EXISTS 是自外向内操作，嵌套查询的复杂度对其存在较大的影响。因此在实际应用中，要考虑子查询的结果集以及子查询的复杂度来决定使用 NOT EXISTS 还是 NOT IN。对于 IN、EXISTS 和 JOIN 等操作，大多数情况下 DB2 优化器都能形成比较一致的最终查询计划。

读者可以在自己的机器上，创建两张表来模拟这个实验以加深理解。

9.4.3 利用子查询进行优化

某些情况下可以将查询中的某一部分逻辑提取出来作为子查询出现，这能够减少扫描的数据量，以及利用索引进行数据检索。请看如下查询：

```
    索引：
```

```
inst1.idx history date on inst1.history(tstmp)
inst1. IDX HISTORY ACCT on inst1.history (ACCT ID)
```

查询：

```
select a.NAME
from inst1.acct  a ,inst1.history h
where a.ACCT ID  =h.ACCT ID  and
          (h.TSTMP > current timestamp - 2 days or
          a.BALANCE  >100 )     ;
```

上面的查询用来选择余额大于 100 或者最近两天有过交易的账户名称。从图 9-4(a)的查询计划中可看出没有任何索引被使用。

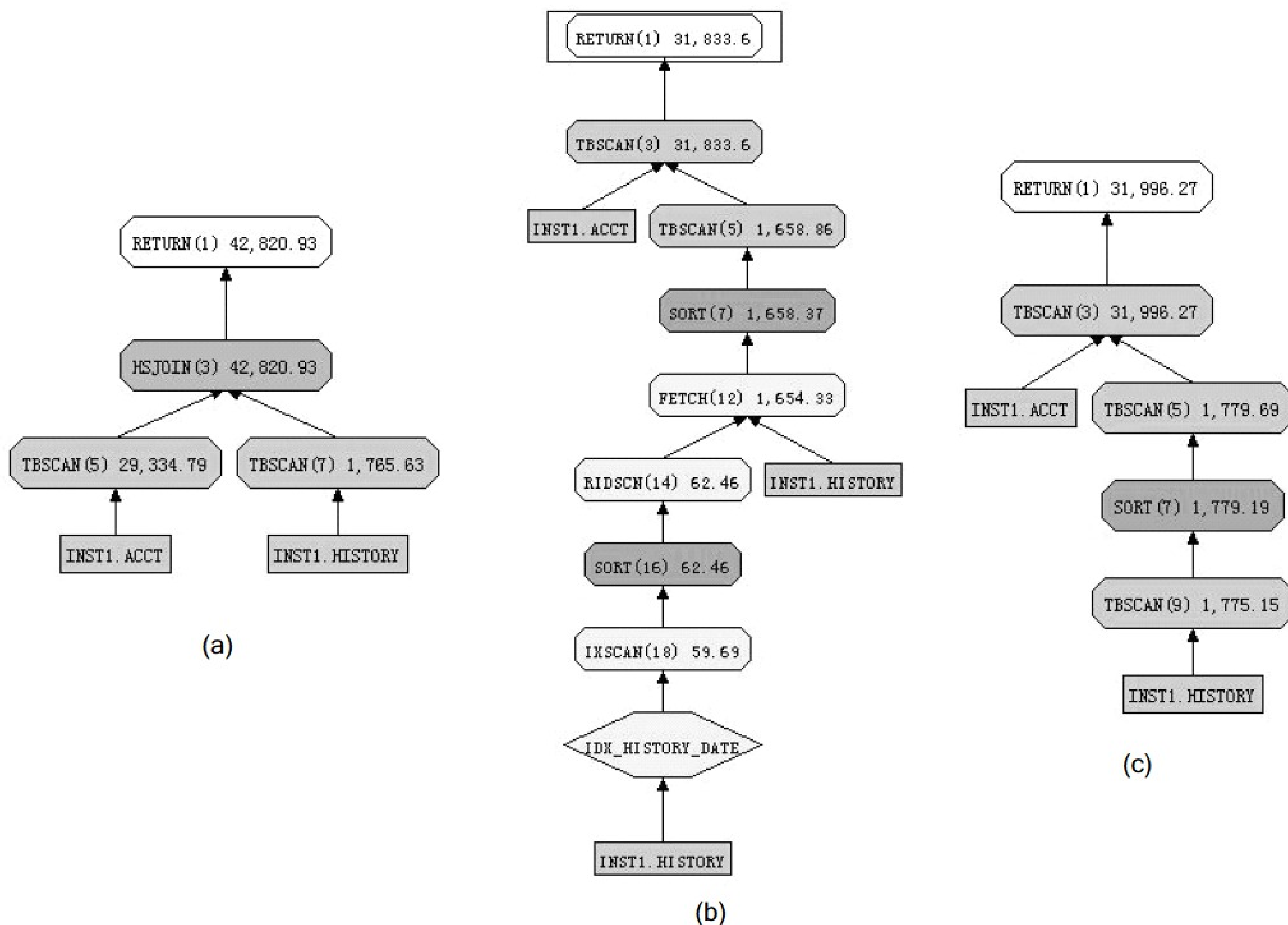


图 9-4　查询计划

使用子查询对上述查询重新改写：

查询：

```
with tmp as(
select ACCT_ID from inst1.history h
```

```
where h.TSTMP > current timestamp - 2 days)
select NAME from inst1.acct  a
where  a.ACCT ID in (select ACCT ID from tmp ) or
      a.BALANCE  >100
;
```

在上面的查询中，我们使用子查询预先限定了要扫描 inst1.history 表中的记录数目，而不是对 inst1.history 表进行全表扫描。同时，在预先限定数据范围的时候，能够利用 inst1.idx_history_date 索引。请看其查询计划，如图 9-4(b)所示。可以看到查询代价有大幅度下降。其实，即使没有 inst1.idx_history_date 索引，修改后的查询也仍然由于前者，因为预先限定了数据的扫描范围，减少了后续连接处理的数据量，请看图 9-4(c)。

9.4.4 调整表连接顺序使 JOIN 最优

一般情况下，DB2 会根据各表的 JOIN 顺序自顶向下顺序处理，因此合理排列各表的连接顺序会提高查询性能。

```
查询:
select a.name,c.point ,h.DELTA
from inst1.acct a
left join   inst1.contact c
on a.acct_id =c.acct_id
join inst1.history h
on h.acct id=a.acct id
where  a.balance >100;
```

上面的查询用来选择出所有余额大于 100 元的账户的所有交易记录和当前积分。此查询会按照连接的顺序先将 inst1.acct 表和 inst1.contact 表进行 LEFT JOIN，然后使用结果集去 JOIN inst1.history 表。由于该查询使用了 LEFT JOIN，因此在生成中间结果集的时候不会有任何记录被过滤掉，中间结果集的记录数目大于等于 inst1.acct 表。了解到 DB2 是如何解释和执行这样的查询后，很自然地我们就会想到将 JOIN 提前。

```
查询:
select a.name,c.point ,h.DELTA
from inst1.acct a
join inst1.history h
on h.acct_id=a.acct_id
left join   inst1.contact c
on a.acct_id =c.acct_id
```

```
    where  a.balance >100;
```

图 9-5(a)和图 9-5(b)分别为上述两个查询的存取计划。在上面的查询中，在形成中间结果集的时候也应用到了 WHERE 语句中的条件，而不是在所有 JOIN 都结束以后才被应用去除记录的。

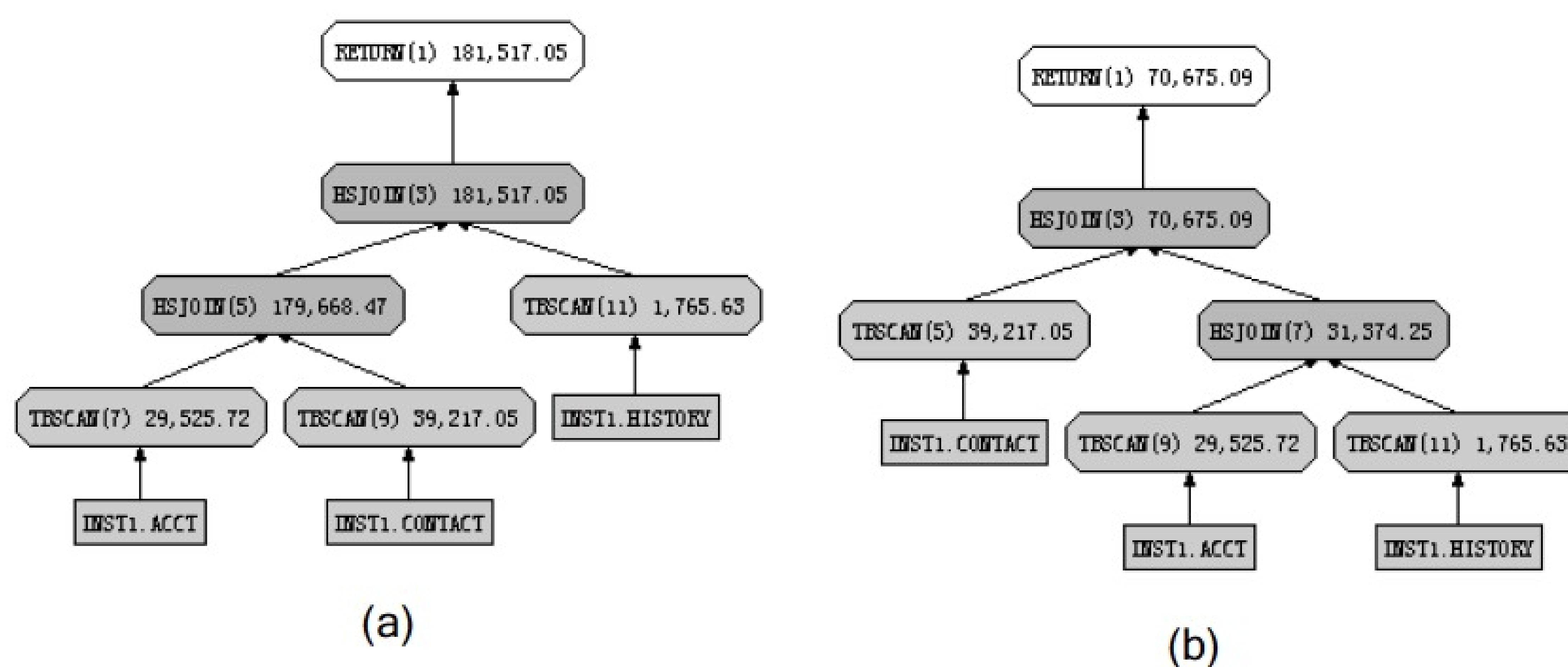


图 9-5　查询计划

我们的原则是保持中间结果集最小，同时考虑中间结果集的数据总量以减少对 JOIN 操作内存的需求。

9.4.5　数据非均匀分布时手工指定选择性

如果数据不是均匀分布的，那么指定选择性就十分有用。SELECTIVITY 指任何一行满足谓词(即为真)的概率。采用具有高度选择性的谓词是可取的。这种谓词将为以后的操作符返回更少的行，从而减少为满足查询所需的 CPU 和 I/O。

例如，对含有 1000000 行的表执行选择性为 0.01(1%)的谓词操作，意味着大约只有 10000 行满足条件，而另外的 990000 行都不满足条件。

如果不是仅仅使用均匀分布的假设，而是人为地使用较低的选择性(例如 0.000001)来“保证”使用那一列上的索引，那么就可以影响优化器。如果预料到表要增长，并且希望确保能够坚持使用某些特定列上的索引，那么这一点就十分有用。如果想阻止 DB2 在某一特定列上使用索引，那么可以将 SELECTIVITY 设为 1。

有了这种技术，使用优化级别 5 (例如 DFT_QUERYOPT=5)就最具有可预测性。而且，首先必须设置注册变量 DB2_SELECTIVITY=YES，然后在使用 SELECTIVITY 子句之前重新启动实例。

可以为下列谓词指定 SELECTIVITY 子句：

- 基本谓词，其中至少有一个表达式包含主机变量/参数标记(基本谓词包括像=、<>、<和<=这样简单的比较符，但是不包括像 IN、BETWEEN 和 IS NULL 这样的东西)。
- 其中的 MATCH 表达式、谓词表达式或换码表达式中包含主机变量/参数标记的 LIKE 谓词。

选择性的值必须是 0 到 1 整个范围内的数值常量(numeric literal)。如果没有指定 SELECTIVITY，那么就会使用默认值。如果 SELECTIVITY 的值为 0.01，那么意味着该谓词将过滤掉除表中所有行的 1%之外的所有其他行。不过应该把提供 SELECTIVITY 看作最后一招。

```
SELECT c1, c2, c3, FROM T1, T2, T3
 WHERE T1.x = T2.x AND
  T2.y = T3.y AND
  T1.x >= ? selectivity 0.000001 AND
  T2.y < ? selectivity 0.5 AND
  T3 = ? selectivity 1
```

9.4.6 使用 UDF 代替查询中的复杂部分

由于 UDF 是预先编译的，性能普遍优于一般的查询，UDF 使用的存取计划一经编译就会相对稳定。我在开发中曾多次发现，使用 UDF 代替查询或视图中的复杂部分会提高几倍甚至几十倍的性能，主要原因是迫使 DB2 使用指定的存取计划来充分利用索引或调整其访问过程(如连接顺序、过滤位置等)。使用 UDF 进行优化的基本思路是：将复杂查询分解为多个部分执行，针对每个部分优化处理，将各部分组合时能够避免存取计划的一些不必要变化，优化整体性能。

```
    查询：
    select * from temp.customer where cust num in
    (select distinct sold to cust num from temp.order where add date > current
timestamp - 2 months
    union
    select distinct cust num from temp.contact
    where add date > current timestamp - 2 months
    )
```

上面这个查询会导致优化器生成比较复杂的查询计划，尤其是在 temp.customer 是比较复杂的视图时。在这种情况下，我们可以通过创建 UDF，将其分步执行：先执行子查询获得 cust_num 值的列表，然后执行最外层的查询。下面的例子是通过 UDF 对上述查询的改写结果：

```
CREATE FUNCTION temp.getCustNum(p date timestamp)
RETURNS
TABLE (cust num CHARACTER(10))
RETURN
select distinct sold to cust num from temp.order
where add date > p date
union
select distinct cust num from temp.contact
where add date > p date;
select * from customer where cust num in (
select cust num from table(temp.getCustNum(current timestamp - 2 months)) tbl
)
```

改写前后的查询代价分别是 445159.31 和 254436.98。当面对比较复杂的查询时，考虑使用 UDF 将其拆分为多步执行常常会带来意想不到的效果。在实际的项目中，如果数据处理和查询调用包含在其他应用程序中(如 UNIX 脚本、Java 程序等)，那么同样可以考虑采用分步数据处理的方式来调用数据库以优化应用性能。

9.4.7　合并多条 SQL 语句到单个 SQL 表达式

跟其他编程语言一样，SQL 语言提供了两类条件构造：过程型(IF 和 CASE 语句)和函数型(CASE 表达式)。在大多数环境中，可使用任何一种构造来表达计算，到底使用哪一种只是喜好问题。但是，使用 CASE 表达式编写的逻辑不但比使用 CASE 或 IF 语句编写的逻辑更紧凑，而且更有效。

请考虑下面的 SQL PL 代码片段：

```
IF (Price <= MaxPrice) THEN
  INSERT INTO tab comp(Id, Val) VALUES(Oid, Price);
ELSE
  INSERT INTO tab comp(Id, Val) VALUES(Oid, MaxPrice);
END IF;
```

IF 子句中的条件仅用于决定将什么值插入 tab_comp.Val 列中。为了避免过程层和数据流层之间的上下文切换，可利用 CASE 表达式将相同的逻辑表示成一个 INSERT 语句：

```
INSERT INTO tab comp(Id, Val)
       VALUES(Oid,
             CASE
                WHEN (Price <= MaxPrice) THEN Price
                ELSE MaxPrice
             END);
```

值得注意的是，CASE 表达式可在任何希望有标量值的上下文中使用。特别地，可在赋值符号的右边使用它们。例如：

```
IF (Name IS NOT NULL) THEN
  SET ProdName = Name;
ELSEIF (NameStr IS NOT NULL) THEN
  SET ProdName = NameStr;
ELSE
  SET ProdName = DefaultName;
END IF;
```

可以改写成：

```
SET ProdName = (CASE
                  WHEN (Name IS NOT NULL) THEN Name
                  WHEN (NameStr IS NOT NULL) THEN NameStr
                  ELSE  DefaultName
                END);
```

实际上，这个特殊的示例有如下更好的解决方案：

```
SET ProdName = COALESCE(Name, NameStr, DefaultName);
```

9.4.8 使用 SQL 一次处理一个集合语义

诸如循环、赋值和游标之类的过程化构造允许我们表达那些只使用 SQL DML 语句是不可能表达的计算。但是，当我们拥有一些可以随意使用的过程语句时，即使我们手头的计算实际上仅使用 SQL DML 语句就可表达，但转换成过程语句还是有风险的。正如我们以前提到的，过程计算的性能与使用 DML 语句表达的同一计算的性能相比会慢几个数量级。请考虑下面的代码片段：

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab sel VALUES (20, v2);
  ELSE
    INSERT INTO tab sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

首先，通过应用前面讨论的转换可以改进循环体：

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  INSERT INTO tab_sel VALUES (CASE
                                WHEN v1 > 20 THEN 20
                                ELSE v1
                              END, v2);
  FETCH cur1 INTO v1, v2;
END WHILE;
```

但是通过进一步观察，我们发现整个代码块可以写成带有 SELECT 子句的 INSERT 语句：

```
INSERT INTO tab_sel (SELECT (CASE
                               WHEN col1 > 20 THEN 20
                               ELSE col1
                             END),
                             col2
                     FROM tab_comp);
```

在原始表述中，SELECT 语句中每行的过程层和数据流层之间都有上下文切换。在最后一个表述中，根本没有上下文切换，并且优化器有机会对整个计算进行全局优化。另一方面，如果每条 INSERT 语句针对的都是不同的表，那么这种简化是不可能的，如下所示：

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_default VALUES (20, v2);
  ELSE
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

但是，这里也可以利用 SQL 的一次处理一个集合(set-at-a-time)特性：

```
INSERT INTO tab_sel (SELECT col1, col2
                     FROM tab_comp
                     WHERE col1 <= 20);
```

```
INSERT INTO tab default (SELECT col1, col2
                    FROM tab comp
                    WHERE col1 > 20);
```

9.4.9　在无副作用的情况下使用 SQL 函数

我们在前面讲过 SQL 过程和 SQL 函数是使用不同技术实现的。SQL 过程中的查询是单独编译的，每个查询都成为包中的一个节。编译是在过程创建时进行的，直到重新创建过程或者直到重新绑定与其相关的包时才重新编译这些查询。

另一方面，SQL 函数中的查询是一起编译的，就好像函数体是查询一样。每当编译一条使用 SQL 函数的语句时，也会对 SQL 函数进行编译。

与 SQL 过程中所发生的情况不同，SQL 函数中的过程语句与数据流语句是在同一层中执行的。因此，每当控制从过程语句流向数据流语句或相反时，并不发生上下文切换。

因为存在这些区别，所以当给定的过程代码段作为函数实现时的执行速度通常比作为过程实现时要快。但是，有一个小问题。函数只能包含那些不会改变数据库状态的语句(例如 INSERT、UPDATE 或 DELETE 语句是不允许的)，并且只允许完整 SQL PL 语言的子集出现在 SQL 函数中(不能是 CALL 语句、游标和条件处理)。

尽管有这些限制，但大多数 SQL 过程都可以在无副作用的情况下转换成 SQL 函数。例如下面的过程：

```
CREATE PROCEDURE GetPrice (IN Vendor CHAR(20),
                      IN Pid INT,
                      OUT price DECIMAL(10,3))
LANGUAGE SQL
BEGIN
  IF Vendor = 'Vendor 1' THEN
      SET price = (SELECT ProdPrice
                FROM V1Table  WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2' THEN
      SET price = (SELECT Price
                FROM V2Table  WHERE Pid = GetPrice.Pid);
  END IF;
END
```

等同于下面的函数：

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), PId INT)
RETURNS DECIMAL(10,3)
LANGUAGE SQL
BEGIN
```

```
    DECLARE price DECIMAL(10,3);
    IF Vendor = 'Vendor 1' THEN
         SET price = (SELECT ProdPrice
                   FROM V1Table  WHERE Id = Pid);
    ELSE IF Vendor = 'Vendor 2' THEN
         SET price = (SELECT Price
                   FROM V2Table  WHERE Pid = GetPrice.Pid);
    END IF;
    RETURN price;
END
```

请注意，尽管使用 CALL 语句来调用过程，但还需要使用 VALUES 语句从命令行调用函数：

```
VALUES (GetPrice('IBM', 324))
```

另一方面，与过程不同的是，您可以在允许表达式的任何上下文中调用函数：

```
SELECT VName FROM Vendors WHERE GetPrice(Vname, Pid) < 100;
SET price =  GetPrice(Vname, Pid);
```

因此，正如本节标题所展示的，当您只是从数据库抽取数据而不执行任何更改时，请考虑使用 SQL 函数而不是使用 SQL 过程。

9.4.10　小结

通过上面的示例，我们发现 SQL 调优并不简单，它包含计划和设计都很好的测试策略、细致的观察和深入的分析。此外，各种平台上的测试结果可能不同。因此，您需要在测试环境中真实反映您的生产环境。但是“一分耕耘，一分收获”，当看到性能获得较大的提高时，我们会感到十分有成就感。我希望大家好好理解上面的示例和分析，这已证明是对您有所帮助的，希望本节能充当你学习 SQL 查询调优的宝典。

9.5　提高应用程序性能

9.5.1　良好的 SQL 编码规则

1) SQL 语句编写应简洁、易读、便于维护。

2) 在每条 SQL SELECT 语句的 SELECT 列表中只提供确实需要检索的那些列。另一种说法就是“不要使用 SELECT *”。简写 SELECT * 表示您要检索正在被访问的表中的所有列。这适用于使用“快捷但不恰当的方式获得的”(quick and dirty)查询，但却是编写

应用程序的坏习惯，因为：

- DB2 表在将来可能需要更改，新增加一些列。SELECT *也会检索那些新的列，而如果没有进行费时的应用更改，您的程序也许无法处理新增加的数据。
- DB2 将为被请求返回的每一列消耗附加资源。如果程序不需要数据，就不会去寻找它们。即使程序需要每一列，最好根据 SQL 语句中的名称来显式地寻找每一列，以便增加清晰度和避免以前犯的错误。

3) 不要寻找您已经知道的东西。这听起来似乎显而易见，但大多数程序员都曾经违反过这条规则。举一个典型的示例，考虑以下 SQL 语句有什么错误：

```
SELECT   EMPNO, LASTNAME, SALARY
FROM     EMP
WHERE    EMPNO = '000010';
```

放弃吗？问题是 EMPNO 已经包含在 SELECT 列表中。您已经知道 EMPNO 将等于值“000010”，因为那就是 WHERE 子句要 DB2 做的事。但在 WHERE 子句中列出了 EMPNO，DB2 还会尽职地检索该列。这会产生附加开销，从而降低性能。

注意：

此处读者不要钻牛角尖，因为实际开发中可能我们确实需要返回这个列。此处的实验仅仅为了说明 SQL 编写时应该注意的原则。

4) 在 SQL 中使用 WHERE 子句过滤数据，而不是在程序中到处使用以进行过滤。这也是开发人员容易犯的错误。在 DB2 将数据返回到程序之前，最好由 DB2 过滤数据。这是因为 DB2 使用附加 I/O 和 CPU 资源来获取每一行数据。传递到程序的行越少，SQL 的效率就越高：

```
SELECT  EMPNO, LASTNAME, SALARY
FROM    EMP
WHERE   SALARY > 50000.00;
```

与只读取所有数据而不使用 WHERE 子句，然后在程序中检查 SALARY 是否大于 50000.00 的做法相比，该 SQL 更好。

5) 确保让重复的语句使用参数标记。使用参数标记查询，可以最小化 DB2 语句的 PREPARE 成本，只需要在 SQL 语句中使用一次参数标记，然后就可以多次重用。参数化 SQL 语句包含了变量，也称作参数(或参数标记)。典型的参数标记查询使用这些参数来代替文字值，因此 WHERE 子句条件可以在运行时更改。通常程序被设计成最终用户可以在运行查询之前提供参数的值。这允许使用查询根据提供给参数的不同值返回不同的结果。使用参数标记，对于 OLTP，编译时间是比较可观的，因此用参数标记替换文字可以避免

重复编译。当使用参数标记时，优化器假设值是均匀分布的，因此如果数据比较偏，那么意味着所选择的访问计划不好，这种情况可考虑对表做分布统计信息更新。

参数标记查询的主要性能好处是优化器可以制定在重复执行语句时能够再次使用的存取路径。与每次 WHERE 子句中需要新值就发出一条全新的 SQL 语句相比，这可以给程序增加很大的性能收益。

9.5.2　提高 SQL 编程性能

1) 要确保在您的应用程序中，在业务逻辑结束后尽快发出 COMMIT 语句。

COMMIT 语句控制事务工作单元。发出 COMMIT 会将自上一条 COMMIT 语句之后的所有工作“永远”记录到数据库中。在发出 COMMIT 之前，可以使用 ROLLBACK 语句回滚工作。当修改数据(使用 INSERT、UPDATE 和 DELETE)但没有发出 COMMIT 时，DB2 将在数据上加锁并保持该锁定，会使其他应用程序在访问被锁住的数据时需要等待或超时。通过在事务完成时发出 COMMIT 语句，并且确保数据是正确的，就释放了锁以提高并发性能。

2) 使用 SELECT ... FOR UPDATE 保护在随后的 UPDATE 语句中可能被更新的那些行。这样一来，选中的所有行上便获得了更新(U)锁。

3) 可以用 SELECT ' FETCH FIRST n ROWS 来限制查询结果集的大小。

读者在构建应用程序时要考虑使用情况。例如，当某个特定查询返回几千行给最终用户时，要慎重处理。对于在程序和最终用户之间的在线交互，很少会用到几百行以上的数据。您可以在 SQL 语句上使用 FETCH FIRST *n* ROWS ONLY 子句来限制返回到查询的数据量。例如，考虑以下查询：

```
SELECT  EMPNO, LASTNAME, SALARY
FROM    EMP
WHERE   SALARY > 10000.00
FETCH FIRST 200 ROWS ONLY;
```

该查询将只返回 200 行。即便有超过 200 行符合条件也没有关系；如果您尝试从查询中 FETCH(访存)超过 200 行，DB2 将用+100 SQLCODE 表明数据结束。当您想要限制返回给程序的数据量时，这种方法很有用。

4) 尽量用相同数据类型的数据进行比较，以避免发生数据转换。

SQL 语言对于数据类型不像 Java 和 C++那样进行严格的数据类型检查，不同种数据间可以进行某些运算，但是在做数据操作时需要数据库进行隐式的类型转换。在大数据量的查询中，由于要对每一个数据项做同样的操作，因而会造成时间和 CPU 处理能力的浪费。

实际应用中通常发生的隐式数据类型转换有：

(1) 字符型到数字型的转换，比如 SELECT '1234' +3 FROM dual 等。

(2) 数字型到字符型的转换，比如 UPDATE DEPT SET EMPNO=5678 等。

上述转换都是隐含发生的，在实际使用中要避免使用不同类型的数据操作。

5) 正确使用 LIKE 通配符。

在应用程序中为了使用方便，对字符型变量进行比较时经常使用 LIKE 运算符进行字符模式的匹配。

需要注意的是：对于 LIKE 运算，如果通配符%出现在字符串的尾部或中间，LIKE 运算符将可以使用索引进行字符串的匹配，否则如果通配符%出现在字符串的开始，LIKE 就必须使用全表扫描的方式去匹配字符串，这将产生较大的系统负载。

一般情况下，为了提高系统的效率，我们希望用户能够在通配符的左端提供较多的数据信息以降低查询的数量。

6) 用 SELECT ' OPTIMIZE FOR *n* ROWS 优化返回时间。

DB2 支持另一条名为 OPTIMIZE FOR *n* ROWS 的子句，该子句不限制要返回给游标的行数，但从性能角度看可能是有帮助的。使用 OPTIMIZE FOR *n* ROWS 子句可告诉 DB2 如何处理 SQL 语句。这样可以使优化器快速地返回 *n* 行，而不是像默认行为那样，最小化整个结果集的代价。此外，如果使用 READ ONLY 子句，这将影响在每个块中返回的行数(一个块中的行数不会大于 *n*)。这不会限制可以取的行数，但是如果要取多于 *n* 行的记录，就可能降低性能。为了使该子句对数据缓冲区有一定的影响，*n* * row size 的值不能超出通信缓冲区的大小(由 DBM CFG RQRIOBLK 或 ASLHEAPSZ 定义)。

例如：

```
SELECT   EMPNO, LASTNAME, SALARY
FROM     EMP
WHERE    SALARY > 10000.00
OPTIMIZE FOR 20 ROWS;
```

这告诉 DB2 尝试尽快访存前 20 行。如果您的应用程序在显示从数据库检索出来的数据行时每次显示 20 行，那么这将非常有用。

7) 使用 SELECT ' FOR READ ONLY(或 FETCH ONLY)表明结果表是只读的。

这意味着不能在随后放置的 UPDATE 或 DELETE 语句中引用游标。这可以帮助提高 FETCH 操作的性能，因为允许 DB2 执行块操作(对于给定的 FETCH 请求返回多行给客户)。

对于只读游标，使用 FOR READ ONLY 子句确保游标无歧义。因此将 FOR READ ONLY 附加到每条 SELECT 语句后面可以使游标成为无歧义的只读游标，从而对 DB2 有所帮助。例如：

```
SELECT   EMPNO, LASTNAME, SALARY
```

```
FROM      EMP
WHERE     SALARY > 10000.00
FOR READ ONLY;
```

9.5.3　改进游标性能

在调整优化现有存储过程的性能时，为消除游标循环而花费的任何时间都是值得的。

如果存储过程中的逻辑确实需要游标，那么要使性能最优，请牢记下面这些内容。

首先，确保不使用高于您所需的隔离级别。隔离级别决定了存储过程读取或更新的行应用的锁定的数量。隔离级别越高，DB2 执行的锁定越多，因此为同一资源而竞争的应用程序之间的并发就越少。例如，使用可重复读(Repeatable Read，RR)隔离级别的过程将形成对其读取的任何行的共享锁，而使用游标稳定性(Cursor Stability，CS)的过程只会锁定任何可更新游标的当前行。可以使用 db2_sqlroutine_prepopts 注册变量来指定 SQL 过程的隔离级别。例如，要将 SQL 过程的隔离级别设置为未提交读(Uncommitted Read，这是最低的隔离级别，用于访问只读数据的存储过程)，请使用下面这条命令：

```
db2set  db2_sqlroutine_prepopts="ISOLATION UR"
```

注意：

要使该设置生效，必须重新启动 DB2 实例。

DB2 中默认的隔离级别是游标稳定性。但是，为了保持应用程序的正确性，有时需要使用可重复读。还需记住一件重要的事情：一旦创建需要可重复读的存储过程，就必须将 DB2_SQLROUTINE_PREPOPTS 重新设置为较低的隔离级别。

有关隔离级别还值得一提的是，DB2 允许我们在单独的查询中覆盖默认的隔离级别，如下所示：

```
DECLARE cur1 CURSOR FOR SELECT col1 FROM tab_comp WITH UR;
```

上面的查询将以隔离级别 UR 进行执行，而不管 DB2_SQLROUTINE_PREPOPTS 中指定的隔离级别为何。

在尝试改进游标性能时需要牢记的一个相关问题是游标的可更新能力。如果游标涉及的行可以使用 INSERT 或 DELETE 语句中的 WHERE CURRENT OF 子句进行更新或删除，那么游标就是可删除的。当游标可删除时，DB2 必须获取行上的互斥锁(与共享锁相对)，并且不能执行行分块(分块游标，blocking_cursor)。行上的互斥锁甚至可以防止其他应用程序读取该行(在互斥锁被释放之前，这些应用程序必须等待，除非它们的隔离级别是 UR)，而行分块通过在操作中检索行块，从而减少了用于游标的数据库管理器开销。

只有不可删除的游标才可以进行行分块。这就是为什么让 DB2 了解将如何使用游标很重要的原因。通过在 SELECT 语句中指定 FOR READ ONLY 子句，可以将游标显式地声明为不可删除，或者通过在 SELECT 语句中使用 FOR UPDATE 子句将其声明为可删除。根据该信息，DB2 将确定是否将行分块用于给定的游标。

默认情况下，对于那些使用 FOR READ ONLY 子句定义的游标，DB2 将始终使用行分块，除非指定了 BLOCKING NO 绑定选项。另一方面，如果使用了 BLOCKING ALL 绑定选项，那么对于模糊(二义)游标(既不是定义成 FOR READ ONLY，也不是定义成 FOR UPDATE 的游标)，DB2 将使用行分块。

简而言之：如果可能，在游标定义中使用 FOR READ ONLY 子句；如果您的存储过程包含模糊游标，那么请使用 BLOCKING ALL 绑定选项。要设置 BLOCKING 绑定选项的值，我们还可以使用 db2_sqlroutine_prepopts 注册变量。例如，要将 SQL 过程的隔离级别设置为未提交读，并将行分块设置为 BLOCKING ALL，请使用下面这条命令：

```
db2set  db2_sqlroutine_prepopts="ISOLATION UR BLOCKING ALL"
```

对于返回大型结果集的过程而言，分块(blocking)特别重要。

通过使用 db2_sqlroutine_prepopts 注册变量，还可以为存储过程指定其他绑定选项。

9.5.4 根据业务逻辑选择最低粒度的隔离级别

为了拥有更好的粒度、更好的性能和并发性，可以在语句级指定业务逻辑允许的最低隔离级别。DB2 支持 UR、CS、RS 和 RR 隔离级别。例如，SELECT * FROM STAFF WITH UR 将使用 Uncommitted Read(最小锁)执行 SELECT 语句。关于锁和隔离级别的详细描述，请参见本书“第 5 章：锁和并发”。

9.5.5 通过 REOPT 绑定选项来提高性能

如果用于输入变量(例如参数标记、宿主变量、全局变量和专用寄存器)的值超出了默认过滤因子估计的预期范围，那么在执行期间 SQL 查询的执行情况可能很差。由于不知道实际数据值的一些方案的默认过滤因子，会估计以下情况：使用实际数据值时，在运行时实际上将返回多少行。

对于占用资源很多的 SQL 语句，查看实际的变量值(字符)而不是参数标记也许对优化器较为有益。实现此目的的一种简单方式就是在代码中使用字符，而不是使用参数标记。但是，这将导致语句缓存发生过多的插入活动，潜在地影响性能和内存使用，因为仅有一条字符值不同的 SQL 语句就会被当作不同的语句。对于包中的静态 SQL(比如 SQL 存储过程)，通过 REOPT ALWAYS 绑定或预编译选项，可以在使用参数标记的同时允许通过值进行优化。对于 DB2 V9 和动态 SQL，也可以通过全局或语句级别的优化配置来指定 REOPT ALWAYS。

REOPT 绑定选项指定是否让 DB2 通过使用宿主变量、参数标记、全局变量和专用寄存器的值来优化访问路径。REOPT 的值分别由 BIND、PREP 和 REBIND 命令的下列自变量指定：

- REOPT NONE：将不使用宿主变量、参数标记、全局变量或专用寄存器的实际值来对包含这些变量的给定 SQL 语句的访问路径进行优化；而是使用这些变量的默认估计值。对此方案进行了高速缓存，并且将来会使用此方案。这是默认行为。
- REOPT ONCE：在第一次执行查询时，将使用宿主变量、参数标记、全局变量或专用寄存器的实际值来优化给定 SQL 语句的访问路径。对此方案进行了高速缓存，并且将来会使用此方案。
- REOPT ALWAYS：将始终使用每次执行时已知道的宿主变量、参数标记、全局变量或专用寄存器的值来编译和重新优化给定 SQL 语句的访问路径。

9.5.6　统计信息、碎片整理和重新绑定

要实时保持数据库的统计信息为最新的，在合适的时间调度 RUNSTATS。定期对表和索引做碎片整理。

当开发嵌入 SQL 程序时，单独的 SQL 查询被编译成程序包中的节。其中，DB2 优化器根据表的统计信息(例如，表大小或某列中数据值出现的相对频率)以及编译查询时可用的索引来选择查询的执行方案。当表经过了重大更改时，让 DB2 再次收集有关这些表的统计信息可能是个好主意。当更新了统计信息时，或者当创建了新的索引时，重新绑定那些与使用表的 SQL 过程相关联的包，以使 DB2 创建使用最新统计信息和索引的方案，这可能也是个好主意。

可以使用 RUNSTATS 命令更新表的统计信息。要重新绑定与 SQL 过程关联的包，可以使用 REBIND_ROUTINE_PACKAGE 内置过程。例如，可以使用下面这条命令来重新绑定过程 MYSCHEMA.MYPROC 的包：

```
CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'MYSCHEMA.MYPROC', 'ANY')
```

其中的'P'表明该包对应于过程，而'ANY'表明 SQL 路径中的任何函数和类型都被当作函数和类型解析。

RUNSTATS、REORG 和 REBIND 的关系如图 9-6 所示。

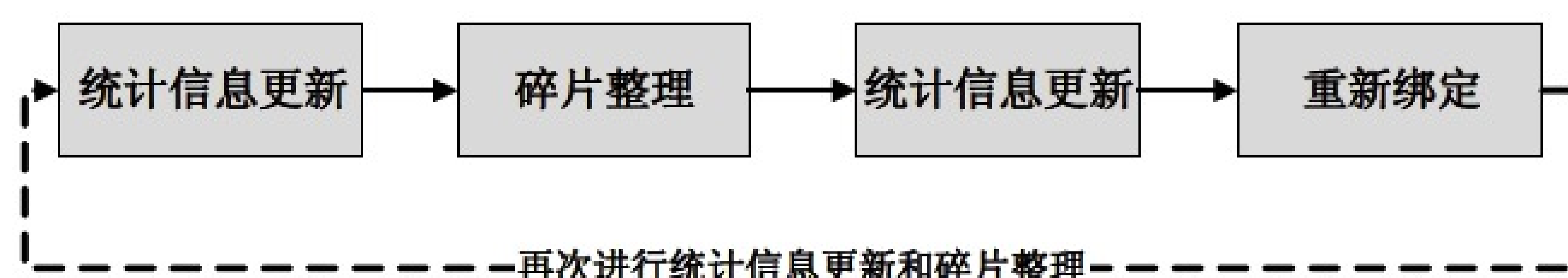


图 9-6　RUNSTATS、REORG 和 REBIND 的关系

9.5.7 避免不必要的排序

排序是数据库中资源消耗比较大的一种操作，我们希望数据库中排序操作的数量能够被尽量减少，同时每个排序的时间能够缩短。在保证业务逻辑正确和允许的情况下，应该：

- 使用 UNION ALL 代替 UNION。
- 添加索引。在表连接的时候使用索引可以避免排序的发生，比如添加合适的索引，可以使连接方式由合并连接(Merge Join)转变为使用索引扫描的嵌套循环连接(Nested Loop Join)。
- 在 DISTINCT、GROUP BY、ORDER BY 子句涉及的列上创建索引。
- 使用较大的 sortheap 和 sheapthres 配置参数。

9.5.8 在 C/S 环境中利用 SQL 存储过程降低网络开销

在 C/S 环境中，通过最小化到客户机的结果集通信量，SQL 存储过程能够降低网络开销，而且存储过程也能够改善静态(预准备的)SQL 的性能。存储过程的其他益处还包括减少客户端处理(通过更多地使用 DB 服务器资源)以及 DB2 的代码管理。

9.5.9 在高并发环境下使用连接池

1) 利用连接池，包括由应用服务器管理的连接池。如果不在应用服务器环境中运行，就使用由应用程序管理的连接。打开和关闭连接的过程开销较大，在高并发环境下，会影响到应用程序或数据库的性能，而使用连接池就可以消除大部分这样的开销。

2) 如果使用了大量的连接，那么请使用 DB2 的连接集中器(connection concentrator)功能。该功能只允许较少的 DB2“后端”连接为应用程序连接服务，从而节省了内存。

3) 有效地使用 JDBC 连接池。首先，确保已经针对列数据类型使用恰当的 setxxx 方法将数据绑定到参数标记。这会避免数据类型转换开销过高和可能的 SQL 数据类型不匹配错误。其次，尽可能避免使用可滚动的结果集设置，因为服务器会实现临时表来支持这种设置。使用这种设置会影响到性能，尤其是在使用大的结果集设置时。

9.5.10 使用 Design Advisor(db2advis)建议索引

一旦发现一条比较消耗资源的 SQL 语句，将该 SQL 语句作为输入使用 Design Advisor 来建议索引。对于经常使用的查询以及大型或复杂的查询，这一点尤其重要。

此外，针对由测试环境压力测试程序填充的动态 SQL 缓存重新运行 Design Advisor。这允许根据实际工作负载和 SQL 语句的执行频率建议索引。确保在 Design Advisor 执行之前运行了 RUNSTATS 统计信息更新。

另一个方法可能需要较多的操作，但是允许利用潜在的(“虚拟的”)索引查看访问计

划。可通过以下步骤实现：

(1) 发出 SQL 语句 SET CURRENT EXPLAIN MODE RECOMMEND INDEXES。

(2) 在同一会话中执行一条 SQL 语句，这会导致用建议的(“虚拟的”)索引填充 ADVISE_INDEX 表。

(3) 执行 SQL 语句 SET CURRENT EXPLAIN MODE EVALUATE INDEXES。

(4) 执行同一条 SQL 语句。现在将使用优化器根据实际和虚拟索引选择的访问计划来填充解释表。

(5) 最后，利用当前和建议的索引，使用诸如 db2exfmt 和 db2expln 这样的解释工具查看访问计划。

9.5.11　提高批量删除、插入和更新速度

大规模的 DELETE/Purging 可以通过使用下面的语句在清除表的全部数据时不做日志记录：

```
ALTER TABLE  ACTIVATE NOT LOGGED INITIALLY WITH EMPTY TABLE
```

由于该操作没有日志记录，如果哪个地方出了错，就不得不将表删除。

另一种高效的方法是使用 LOAD 实用程序(带上 Replace 选项)和一个空文件，同样可以在不记录日志的情况下将表的数据库清除。在执行清除数据的 LOAD 操作之前，最好使用 EXPORT 工具将数据备份出来。

为了提高插入和更新数据的速度，同样可以通过不记录日志的方式。当然，由于不记录日志会造成无法恢复的问题，因此使用之前要谨慎考虑。

使用如下语句可以实现不记录日志：

```
ALTER TABLE table-name ACTIVATE NOT LOGGED INITIALLY
```

注意：

上面的语句必须与执行数据操作的 SQL 语句在同一事务中。而且不记录日志设置的影响在最近一个事务提交之后就会自动失效。

9.5.12　提高插入性能

在将数据插入到表中之前，插入搜索算法将检查可用空间控制记录(FSCR)，进而查找空间足以存储新数据的页。

但是，即使 FSCR 指示某页的可用空间足够，该空间也可能因为已被另一个事务中未落实的删除操作保留而不可用。

db2maxfscrsearch 注册变量指定将记录添加到表时要搜索的 FSCR 数目。默认情况是搜索 5 个 FSCR。修改此值使您能够在插入速度与空间复用之间进行平衡。使用较大的值将优化空间复用。使用较小的值将优化插入速度。将值设置为 -1 表示强制数据库管理器搜索所有 FSCR。如果搜索 FSCR 时找不到足够的空间，那么数据将被追加到表的末尾。

ALTER TABLE 语句的 APPEND ON 选项指定将追加表数据，并指定不保留关于页中可用空间的信息。这样的表不能带有集群索引。对于只增大不减小的表，此选项能够提高性能。

如果已对该表定义集群索引，那么数据库管理器会尝试将记录插入到其他具有类似索引键值的记录所在的页。如果该页上没有空间，那么将考虑周围的页。如果那些页不合适，那么将搜索 FSCR，如上所述。但是，在这种情况下，将使用“最差匹配”方法来代替“首先匹配”方法。最差匹配方法往往选择包含更多可用空间的页。此方法将为具有类似键值的行建立新的集群区。

如果已对表定义集群索引，那么在装入或重组该表之前，请使用 ALTER TABLE 语句的 PCTFREE 子句。PCTFREE 子句指定执行装入或重组操作后应该在数据页中保留的可用空间所占的百分比。这将提高集群索引操作能在适当的页中找到可用空间的可能性。

9.5.13 高效的 SELECT 语句

因为 SQL 是一种灵活的高级语言，所以您可以编写几种不同的 SELECT 语句来检索同一数据。但是，对于不同的语句形式和不同的优化类，性能可能相差很大。

请考虑下列有关创建高效 SELECT 语句的准则：

- 仅指定需要的列。使用星号(*)指定所有列将产生不必要的处理。
- 使用谓词将结果集限制为仅包括所需的行。
- 如果需要的行数远远小于可能返回的总行数，那么请指定 OPTIMIZE FOR 子句。此子句将影响对访问方案的选择以及在通信缓冲区中分块的行数。
- 要利用行分块方法并提高性能，请指定 FOR READ ONLY 或 FOR FETCH ONLY 子句。并且因为不会对检索到的行挂起互斥锁定，所以并行性也有所改进。此外，还可能会发生其他的查询重写。同样，通过指定这些子句以及 BLOCKING ALL 绑定选项，可以提高对联合数据库系统中昵称运行的查询的性能。
- 对于将与定位型更新操作配合使用的游标而言，指定 FOR UPDATE OF 子句将使数据库管理器能够选择更合适的初始锁定级别并避免发生潜在的死锁。注意，FOR UPDATE 游标无法利用行分块方法。

- 对于将与搜索型更新操作配合使用的游标而言，指定 FOR READ ONLY 和 USE AND KEEP UPDATE LOCKS 子句将对受影响的行强制挂起 U 锁定，从而避免死锁并仍允许进行行分块。
- 尽可能避免进行数字数据类型转换。在比较值时，请尝试使用具有相同数据类型的项。如果需要执行转换，那么会由于精度受限而导致结果不准确，并且会由于进行运行时转换而导致性能下降。

  在有可能的时候，请使用下列数据类型：
  - 对于较短的列，尽量使用字符而不是可变字符。
  - 尽量使用整数，而不是浮点数、小数或 DECFLOAT。
  - 尽量使用 DECFLOAT，而不是小数。
  - 尽量使用日期时间，而不是字符。
  - 尽量使用数字，而不是字符。
- 为了减小发生排序操作的可能性，请省略 DISTINCT 或 ORDER BY 之类的子句(如果此类操作不是必需的)。
- 要检查表中是否存在行，请选择单一行。请打开游标并访存一行，或执行单行 SELECT INTO 操作。记住，如果找到多行，那么需要检查是否发生 SQLCODE -811 错误。

  除非您知道表非常小，否则不要使用以下语句来检查非零值：

  ```
  select count(*) from <table-name>
  ```

  对于大型表，对所有行计数将影响性能。
- 如果更新活动较少且表较大，那么请对谓词中频繁使用的列定义索引。
- 如果同一列出现在多个谓词中，那么请考虑使用 IN 列表。对于配合主变量使用的大型 IN 列表，循环部分主变量可能会提高性能。

下列建议只适用于访问多个表的 SELECT 语句：

- 使用连接谓词来连接表。连接谓词是指连接中不同表的两个列之间的比较。
- 对连接谓词中的列定义索引，以便更高效地处理连接。对于包含访问多个表的 SELECT 语句的 UPDATE 和 DELETE 语句而言，索引也有助于提高性能。
- 有可能时，避免使用包含连接谓词的 OR 子句或表达式。
- 在分区数据库环境中，建议根据连接列对连接的表进行分区。

9.6 高性能 SQL 语句注意事项

9.6.1 避免在搜索条件中使用复杂的表达式

避免在搜索条件中使用复杂的表达式，这些表达式将导致优化器无法使用目录统计信息来估算精确的选择性。

表达式还可能限制对可用于应用谓词的访问方案的选择。在优化的查询重写阶段，优化器可以重写许多表达式以便估算精确的选择性；但无法处理所有的可能性。

9.6.2 将 OPTIMIZE FOR *n* ROWS 子句与 FETCH FIRST *n* ROWS ONLY 子句配合使用

OPTIMIZE FOR *n* ROWS 子句通知优化器，应用程序计划只检索 *n* 行，但是查询将返回完整的结果集。FETCH FIRST *n* ROWS ONLY 子句指示查询应该只返回 *n* 行。

对外子查询指定 FETCH FIRST *n* ROWS ONLY 之后，DB2 数据服务器不会自动采用 OPTIMIZE FOR *n* ROWS。请尝试同时指定 OPTIMIZE FOR *n* ROWS 和 FETCH FIRST *n* ROWS ONLY，以鼓励使用直接从所引用的表返回行而不首先执行缓存操作(例如插入到临时表、执行排序或插入到散列连接散列表)的查询访问方案。

应用程序如果指定了 OPTIMIZE FOR *n* ROWS 以鼓励使用避免缓存操作的查询访问方案，但却检索整个结果集，那么性能可能会欠佳。这是因为，返回前 *n* 行的速度最快的查询访问方案在检索整个结果集时可能不是最好的查询访问方案。

9.6.3 避免使用冗余的谓词

避免使用冗余的谓词，当它们跨不同的表出现时尤其如此。在某些情况下，优化器无法检测谓词是否冗余。这可能导致低估基数。

例如，在 SAP 商业智能(BI)应用程序中，将带有事实表和维表的雪花模式用作查询优化数据结构。在某些情况下，对事实表和维表定义了冗余的时间特征列(用于月份的 SID_0CALMONTH 或用于年份的 SID_0FISCPER)。

SAP BI 联机分析处理(OLAP)处理器将对维表和事实表的时间特征列生成冗余的谓词。这些冗余的谓词可能会导致查询运行时间延长。

9.6.4 节提供了一个示例，在那个示例中，SAP BI 查询的 WHERE 条件中定义了两个冗余的谓词。对时间维(DT)和事实(F)表定义了完全相同的谓词：

```
        AND (     "DT"."SID 0CALMONTH" = 199605
             AND "F". "SID_0CALMONTH" = 199605
```

```
            OR "DT"."SID_0CALMONTH" = 199705
           AND "F". "SID_0CALMONTH" = 199705 )
    AND NOT (     "DT"."SID_0CALMONTH" = 199803
           AND "F". "SID_0CALMONTH" = 199803 )
```

DB2 优化器不会将这些谓词识别为等同，而是将它们视为相互独立。这将导致低估基数、查询访问方案欠佳以及查询运行时间延长。

因此，特定于 DB2 数据库平台的软件层将除去冗余的谓词。上述谓词将转换为如下所示的谓词，只保留了应用于事实表列 SID_0CALMONTH 的谓词：

```
      AND (     "F". "SID_0CALMONTH" = 199605
           OR "F". "SID_0CALMONTH" = 199705 )
      AND NOT (     "F". "SID_0CALMONTH" = 199803 )
```

请应用 SAP 注意事项 957070 和 1144883 中的指示信息以除去冗余的谓词。

9.6.4　避免使用多个带有 DISTINCT 关键字的聚集操作

请避免使用在同一子查询中执行多次 DISTINCT 聚集操作的查询，这些查询的运行成本高昂。

请考虑以下示例：

```
   SELECT SUM(DISTINCT REBATE), AVG(DISTINCT DISCOUNT)
     FROM DAILY_SALES
     GROUP BY PROD_KEY
```

要确定相异 REBATE 值和相异 DISCOUNT 值的集合，可能需要对来自 PROD_KEY 表的输入流进行两次排序。此查询的查询访问方案可能类似于图 9-7 所示。

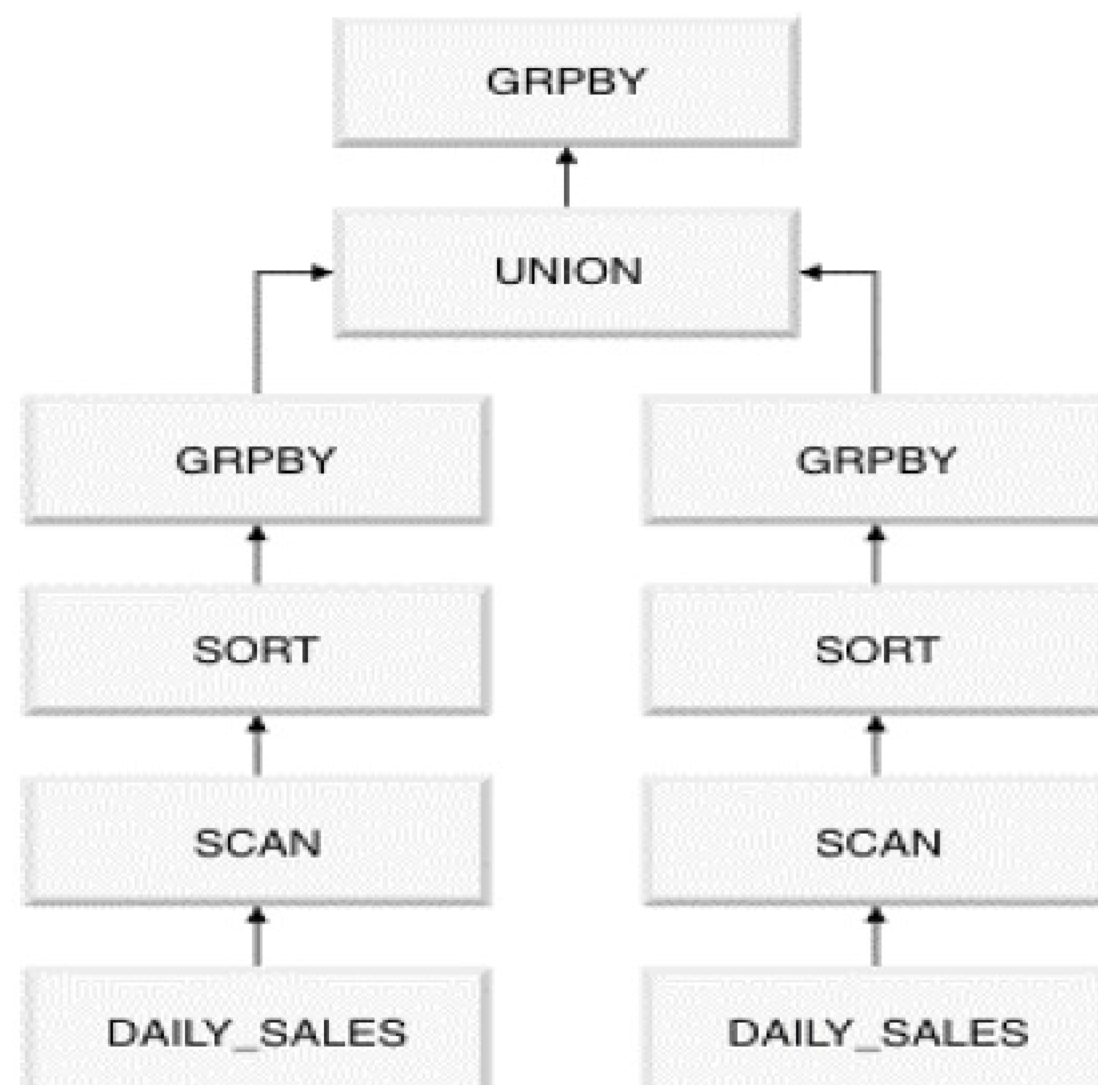


图 9-7　查询访问方案

优化器将原始查询重写为不同的聚集并对每个聚集指定 DISTINCT 关键字，然后使用 UNION 关键字对多个聚集进行组合。在内部重写的语句如下：

```
SELECT Q8.MAXC0, (Q8.MAXC1 / Q8.MAXC2)
FROM
  (SELECT MAX(Q7.C0) AS MAXC0, MAX(Q7.C1) AS MAXC1, MAX(Q7.C2) AS MAXC2
    FROM
      (SELECT SUM(DISTINCT Q2.REBATE) AS C0, CAST(NULL AS INTEGER) AS C1,
        0 AS C2, Q2.PROD_KEY
      FROM
        (SELECT Q1.PROD_KEY, Q1.REBATE
          FROM DB2USER.DAILY_SALES AS Q1) AS Q2
      GROUP BY Q2.PROD_KEY
      UNION ALL
      SELECT CAST(NULL AS INTEGER) AS C0, SUM(DISTINCT Q5.DISCOUNT) AS C1,
          COUNT(DISTINCT Q5.DISCOUNT) AS C2, Q5.PROD_KEY
        FROM
          (SELECT Q4.PROD_KEY, Q4.DISCOUNT
            FROM DB2USER.DAILY_SALES AS Q4) AS Q5
        GROUP BY Q5.PROD_KEY) AS Q7
      GROUP BY Q7.PROD_KEY) AS Q8
```

如果无法避免使用多个 DISTINCT 聚集，那么请考虑将 db2_extended_optimization 注册变量与 ENHANCED_MULTIPLE_DISTINCT 选项配合使用。此选项将导致读一次以多个相异聚集为目标的输入流，然后对 UNION 的每个分支重复使用该输入流。此选项可以提高这些类型的查询的性能。其中，处理器数与数据库分区数的比率较低(例如，比率小于或等于 1)。在不包含对称多处理器(SMP)的分区数据库环境中，应该使用此设置。此优化扩展功能并不能在所有环境中提高查询性能。您应执行测试以确定各个查询的性能提高情况。

9.6.5 避免连接列之间数据类型不匹配

在某些情况下，数据类型不匹配将导致无法使用散列连接。与其他连接方法相比，散列连接对连接谓词有一些额外的限制。尤其是，连接列的数据类型必须完全相同。例如，如果一个连接列是 FLOAT，而另一个是 REAL，那么不支持散列连接。另外，如果连接列的数据类型是 CHAR、GRAPHIC、DECIMAL 或 DECFLOAT，那么长度必须相同。

9.6.6 避免对表达式使用连接谓词

如果对表达式使用连接谓词，那么连接方法只能是嵌套循环连接，并且估算的基数有可能不准确。包含表达式的连接的一些示例如下所示：

```
WHERE SALES.PRICE * SALES.DISCOUNT = TRANS.FINAL PRICE
WHERE UPPER(CUST.LASTNAME) = TRANS.NAME
```

9.6.7　避免在谓词中使用空操作表达式来更改优化器估算

格式为 COALESCE(X, X) = X 的空操作 coalesce()谓词将导致任何使用它的查询的规划发生估算错误。目前，DB2 查询编译器无法详细分析该谓词并确定所有行都确实满足它的要求。

因此，该谓词将人为减少来自某个查询规划部分的估算行数。行估算值减小通常会导致该查询规范中其余部分的行和成本估算值减小，有时还会导致选择另一个方案，这是因为不同候选方案之间的相对估算值已更改。

为何这种空操作谓词有时能够提高查询性能？添加空操作 coalesce()谓词将引入一个错误，该错误将屏蔽其他某些导致性能无法达到最佳水平的内容。

某些性能增强工具执行的是强制测试：工具反复地将该谓词引入到查询中的不同位置，从而对不同的列执行操作，以便尝试找到能够通过引入错误在无意中发现性能更好的方案的情况。对于在查询中手动编码“空操作”谓词的查询开发者而言，情况亦如此。通常，开发者对数据有一定程度的了解，这有助于确定谓词的位置。

使用此方法来提高查询性能是一种短期的解决方案，这无法解决根本原因，并且可能会造成以下影响：

- 可能在能够提高性能的领域未显现。
- 无法保证此变通方法能够永久提高性能，这是因为，DB2 查询编译器最终可能能够更好地处理谓词，其他随机因素也可能会对其产生影响。
- 其他查询也可能会受同一根本原因影响，这通常会导致系统性能有所下降。

如果您已遵循最佳实践建议，但相信仍未实现最佳性能，那么可以为 DB2 优化器提供明确的优化准则，而不是引入空操作谓词。

9.6.8　确保查询符合星型模式连接的必需条件

对于星型模式，优化器将考虑两种专用的连接方法——星型连接或集线器连接，它们有助于显著提高性能。

但是，查询必须符合以下条件：

- 对于每个查询块
  - 必须至少连接 3 个不同的表
  - 所有连接谓词都必须是等式谓词
  - 不能存在子查询

- 在各个表之间或查询块外部不能存在相关性或依赖性
- 对于索引与(AND)运算，不能存在不确定函数，这是因为索引必须应用事实表谓词以便于进行半连接
- 事实表
  - 是查询块中最大的表
  - 至少包含 10000 行
  - 被认为仅仅是一个表
  - 必须至少连接至两个维表或称为“雪花”的组
- 维表
  - 不是事实表
  - 可以逐个连接到事实表或者在“雪花”中进行连接
- 维表或“雪花”
  - 必须对事实表进行过滤(过滤基于优化器的估算值)
  - 必须存在用于事实表并且使用事实表索引中的前导列的连接谓词。必须符合此条件才能考虑使用星型连接或集线器连接，尽管集线器连接只需要使用单一事实表索引

表示左外连接或右外连接的查询块只能引用两个表，因此星型模式连接不符合条件。

不需要显式地声明引用完整性即可使优化器识别星型模式连接。

9.6.9 避免使用非等式连接谓词

因为连接方法只能是嵌套循环连接，所以应该避免使用比较运算符不是等式连接谓词。

另外，优化器可能无法准确计算连接谓词的选择性估算值。但是，并非始终能够避免使用非等式连接谓词。有必要使用非等式连接谓词时，确保存在基于任何一个表的适当索引，这是因为连接谓词将应用于嵌套循环连接的内表。

非等式连接谓词的一个常见示例是，必须对星型模式中的维数据进行版本化才能准确地反映某个维在不同时间点的状态。这通常称为缓慢变化的维。其中一种缓慢变化的维涉及包括每个维行的有效开始日期和结束日期。事实表与维表之间的连接除了要求根据维主键进行连接以外，还要求检查与事实相关联的日期是否在维的开始日期与结束日期之间。这通常称为第 6 类缓慢变化的维。通过某个事实事务日期向后连接到事实表以便进一步限定维版本的范围连接成本很高。例如：

```
SELECT...
  FROM PRODUCT P, SALES F
  WHERE
```

```
    P.PROD_KEY = F.PROD_KEY AND
    F.SALE_DATE BETWEEN P.START_DATE AND
    P.END_DATE
```

在这种情况下，确保存在基于(F.PROD_KEY, F.SALE_DATE)的索引。

请考虑创建统计视图以帮助优化器为此方案计算更好的选择性估算值。例如：

```
CREATE STATISTICAL VIEW V_PROD_FACT AS
  SELECT P.*
    FROM PRODUCT P, SALES F
    WHERE
      P.PROD_KEY = F.PROD_KEY AND
      F.SALE_DATE BETWEEN P.START_DATE AND
      P.END_DATE
ALTER VIEW V_PROD_FACT ENABLE QUERY OPTIMIZATION
RUNSTATS ON TABLE DB2USER.V_PROD_FACT WITH DISTRIBUTION
```

如果查询块中存在任何非等式连接谓词，那么不考虑专用的星型模式连接，例如使用索引与(AND)运算的星型连接和集线器连接。

9.6.10　避免使用不必要的外连接

某些查询的语义需要外连接(左连接、右连接或全连接)。但是，如果查询语义不需要外连接，并且该查询正被用于处理不一致的数据，那么它最适合于处理数据不一致问题的根本原因。

例如，在具有星型模式的数据集市中，事实表可能包含事务的行，但由于数据一致性问题，某些维没有匹配的父维。发生此问题的原因可能是，抽取、变换和装入(ETL)过程由于某种原因而未能对某些业务键进行协调。在此方案中，事实表行以左外连接方式与维连接，确保它们即使没有父代也会被返回。例如：

```
SELECT...
  FROM DAILY_SALES F
    LEFT OUTER JOIN CUSTOMER C ON F.CUST_KEY = C.CUST_KEY
    LEFT OUTER JOIN STORE S ON F.STORE_KEY = S.STORE_KEY
  WHERE
    C.CUST_NAME = 'SMITH'
```

左外连接也会导致无法进行多项优化，其中包括使用专用的星型模式连接访问方法。但是在某些情况下，查询优化器可以自动地将左外连接重写成内连接。在本例中，由于谓词 C.CUST_NAME = 'SMITH' 将除去任何在此列中包含空值的行，从而使左外连接在语义上非必需，所以 CUSTOMER 与 DAILY_SALES 之间的左外连接可以转换为内连接。因此，

由于存在外连接而无法进行某些优化可能不会对所有查询产生负面影响。但是，了解这些限制并避免使用外连接(除非绝对有需要使用外连接)至关重要。

9.6.11 使用参数标记来缩短动态查询的编译时间

DB2 数据服务器可以将访问节和语句文本存储在动态语句高速缓存中，从而避免重新编译先前已运行的动态 SQL 语句。

对此语句发出的后续准备请求将尝试在动态语句高速缓存中查找访问节，从而避免进行编译。但是，仅在谓词中使用的字面值方面有所不同的语句不匹配。例如，下面这两条语句在动态语句高速缓存中被认为不相同：

```
SELECT AGE FROM EMPLOYEE WHERE EMP_ID = 26790
SELECT AGE FROM EMPLOYEE WHERE EMP_ID = 77543
```

即使频繁运行相对简单的 SQL 语句，也会由于语句编译工作而导致系统 CPU 使用率过高。如果系统遇到此类性能问题，那么请考虑更改应用程序，以便使用参数标记将谓词值传递给 DB2 编译器，而不要显式地将它们包括在 SQL 语句中。但是，对于在谓词中使用了参数标记的复杂查询而言，此访问方案可能并非最佳。

9.6.12 使用约束来提高查询优化程度

请考虑定义唯一约束、检查约束和引用完整性约束。这些约束将提供语义信息，这些信息使 DB2 优化器能够重写查询以消除连接、通过连接下推聚集、通过连接下推 FETCH FIRST *n* ROWS、除去不必要的 DISTINCT 操作以及执行许多其他优化。

当应用程序本身能够确保关系时，还可以使用参考约束来代替检查约束和引用完整性约束。在这种情况下，可以进行相同的优化。在插入、更新或删除行时，数据库管理器强制实施的约束可能会引起大量系统开销，更新大量具有引用完整性约束的行时尤其如此。如果应用程序在更新行之前已验证信息，那么使用参考约束可能比使用常规约束效率高。

例如，假定有两个表 DAILY_SALES 和 CUSTOMER。CUSTOMER 表中的每一行都有唯一的客户键(CUST_KEY)。DAILY_SALES 包含 CUST_KEY 列，并且每一行都引用 CUSTOMER 表中的客户键。您可以创建引用完整性约束以表示 CUSTOMER 与 DAILY_SALES 之间的这种 1:N 关系。如果应用程序强制实施此关系，那么可以将该约束定义为参考约束。于是，以下查询可以避免在 CUSTOMER 与 DAILY_SALES 之间执行连接，这是因为不会从 CUSTOMER 中检索任何列，并且 DAILY_SALES 中的每一行都将在 CUSTOMER 中找到匹配项。查询优化器将自动除去连接。

```
SELECT AMT_SOLD, SALE_PRICE, PROD_DESC
  FROM DAILY_SALES, PRODUCT, CUSTOMER
```

```
    WHERE
      DAILY_SALES.PROD_KEY = PRODUCT.PRODKEY AND
      DAILY_SALES.CUST_KEY = CUSTOMER.CUST_KEY
```

应用程序必须强制实施参考约束，否则查询可能会返回不正确的结果。在此例中，如果 DAILY_SALES 中的任何行在 CUSTOMER 表中没有相应的客户键，那么此查询将不正确地返回那些行。

9.7　本章小结

本章我们讨论了 SQL 语句调优的相关内容，包括 SQL 语句的工作机制、调优工具以及调优时的注意事项。掌握了这些知识后，我们在今后的 SQL 使用过程中就可以利用本章提到的工具和方法来提高 SQL 语句的效率，同时也为今后日常的 SQL 调优工作打下良好的基础。

第10章 DB2 集群调优

DB2 在 V9.7 的同时期发行了特殊的集群版本 V9.8，也就是 pureScale 集群。从 V10 开始，pureScale 作为 DB2 的可选功能组件，和 DPF 一样，包含在企业版和高级企业版的软件许可中。本章所说的 DB2 集群，就是专指 pureScale 特性。

DB2 pureScale 集群就像 Oracle RAC 技术，是面向 OLTP 的高可用、可横向扩展的集群解决方案。DB2 pureScale 集群实现的技术和 Oracle RAC 完全不同，虽然都是共享存储架构，但在内部的通信机制上，处理机制有很大的区别。所以我们要先了解 DB2 集群的内部工作机制，才能理解在应用 DB2 集群过程中会遇到什么样的特殊问题。

本章主要讲解如下内容：

- DB2 集群介绍
- DB2 集群特殊参数解析
- DB2 集群性能监控
- DB2 集群设计调优
- 同城双活集群介绍
- 同城双活集群调优

10.1 DB2 集群介绍

DB2 pureScale 是同时兼备高扩展性和高可用性的数据库集群，同时对于应用是透明

的。无论连接到集群内哪个成员节点，连接到的都是同一个数据库。图 10-1 总结了 DB2 集群的技术架构。

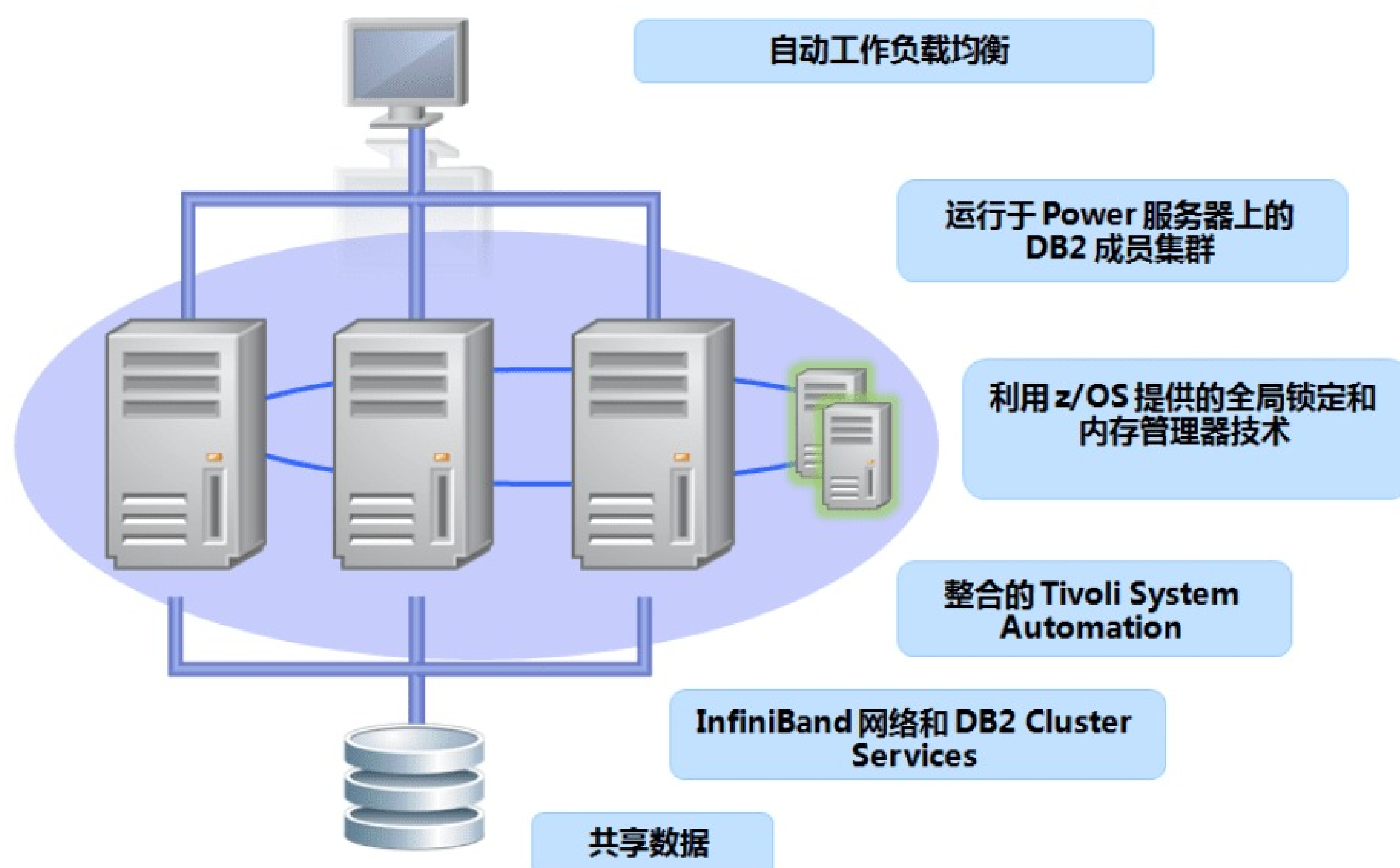


图 10-1　DB2 pureScale 集群架构

首先，DB2 集群是共享存储架构，每个数据库处理节点(成员)都可以直接操作磁盘数据。共享存储使用的是 IBM GPFS 集群文件系统。这个集群文件系统是整个 DB2 集群的底层基础，被 DB2 集群所管理，而整个 DB2 集群的管理软件核心是 Tivoli System Automation 集群软件。TSA 负责监控整个集群里面资源的状态，包括存储、网络、进程等，并且基于这些资源的状态自动化相应的行为。例如 TSA 监控到某个节点的网卡宕机，就会自动关闭当前节点的资源，并漂移到其他健康节点。

在共享存储的基础上，集群内部的通信对响应时间要求很高。DB2 pureScale 集群支持 RDMA 协议和 TCP/IP 协议。RDMA 协议相对速度更快，资源消耗更小，在高性能运算中建议使用。而 TCP/IP 协议部署成本低，适合于高可用场景。为了协调各节点一起服务同一个数据库，DB2 集群内部引入了 CF 功能，这个功能推荐部署为单独的节点。CF 节点主要提供全局锁定和内存管理器技术，继承于 IBM z/OS 系统。现在 DB2 集群支持 Power 服务器和 X86 服务器两种开放式平台。

DB2 集群数据库处理节点多，对于上层应用来说，客户端可选的连接方式也比较齐全。主要运用的客户端连接方式有两种：自动工作负载均衡和客户端偏好设置。 如果使用自动工作负载均衡，那么数据库服务器会不断给客户端反馈当前的机器节点负载列表，客户端

会基于此列表分发事务；而如果选择客户端偏好设置，那么客户端会一直连接首选的数据库成员节点，只有在这个偏好的节点连接不上时，才会自动连接预先设置的节点列表中的下一个节点。这两种方式各有利弊。

10.2　DB2 集群参数解析

如果想要运用好 DB2 pureScale 集群，首先要了解 CF 节点的功能。前面说了，加入 CF 节点是为了协调处理所有成员节点的工作。其中最核心的功能是全局锁管理器(GLM，Global Lock Manager)和组缓冲池(GBP，Group Buffer Pool)。

10.2.1　组缓冲池

在 DB2 pureScale 环境中，集群高速缓存设施提供了由所有成员共享的组缓冲池(GBP)。每个成员还可管理自身的一组本地缓冲池(LBP)。本地缓冲池其实就相当于单机版数据库的缓冲池。GBP 是支持所有 DB2 页大小并且所有成员都可使用的单个缓冲池。成员会将页高速缓存在它们自己的 LBP 中，并使用 GBP 来维护成员之间的页一致性。每个成员上可存在不同页大小的 LBP(例如，4KB、8KB、16KB 或 32KB)。

图 10-2 演示了 GBP 与 LBP 的关系。

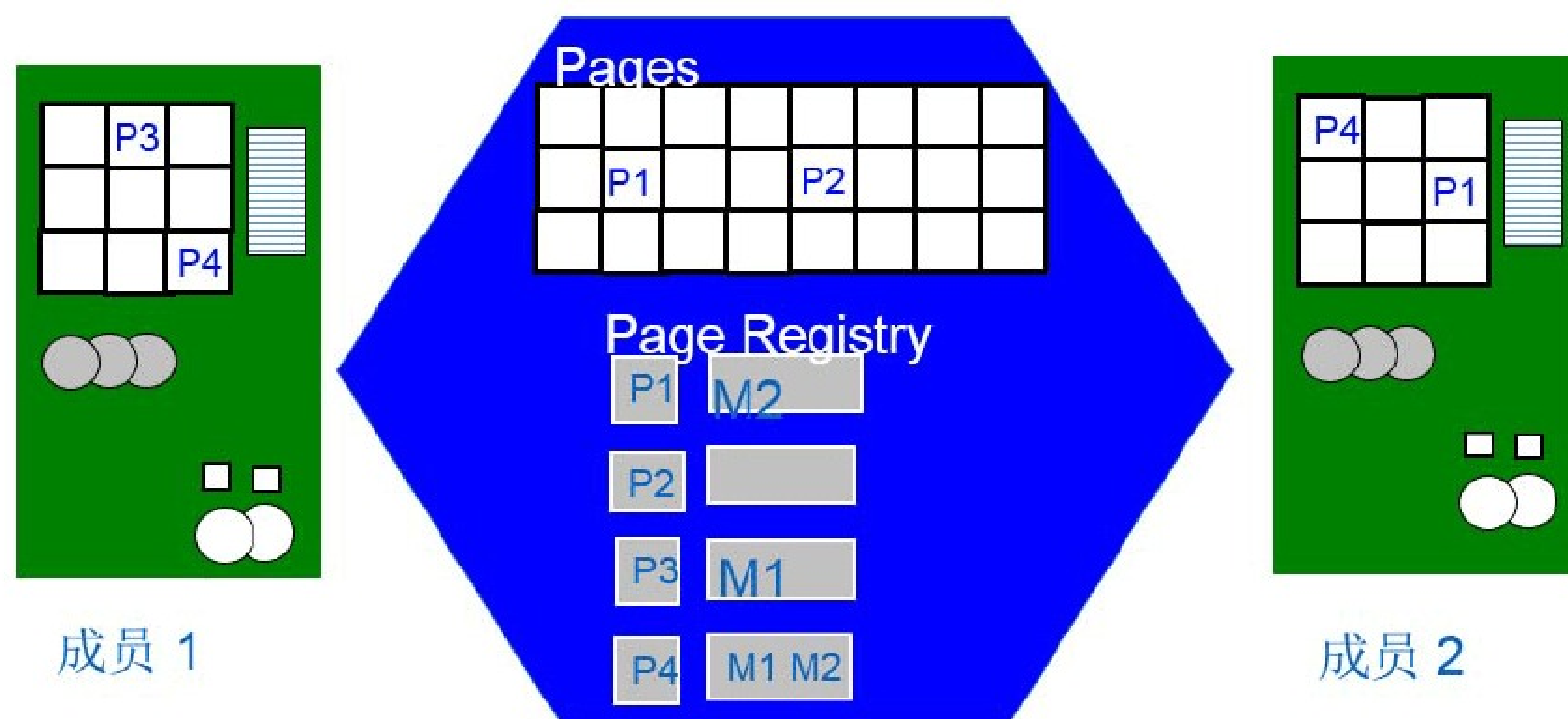


图 10-2　GBP 功能示意

GBP 存储两种类型的信息：目录条目和数据元素。目录条目存储有关缓冲池页的元数据信息，数据元素存储页数据。目录条目与数据元素的比例由 DB2 自动调整。

因为每个成员上有 LBP 并且存在由所有成员共享的 GBP，所以同一页的多个副本可

存在于多个缓冲池中。用于访问页、更改页以及将更改传播至其他成员的全局并行性和相干性控制由 DB2 缓冲池服务处理。此服务还会处理缓冲池中的 I/O 数据，包括将 GBP 中的页写至磁盘。

但是为了控制缓冲池数据的有效性，协调不同成员节点之间的事务处理和数据共享，在 DB2 pureScale 集群中引入了一种新的锁定 P-lock。P-lock 控制 DB2 pureScale 环境中缓冲池页的访问以更新和读取某个页版本。与由特别事务拥有的逻辑锁定(例如行锁定或表锁定)不同，用于控制缓冲池页访问的 P-lock 由集群的成员所有。系统使用了以下 P-lock：

- 要更新页，成员必须以 X 方式持有 P-lock。
- 要读取页的最新版本，成员必须以 S 方式持有 P-lock。

要读取此页的一致版本(但不一定是最新版本)，不需要任何 P-lock。DB2 内部决定访问页时使用哪种类型的读取。

以下协议规则协调多个页副本的相干性：

- 成员将某页访存到其 LBP 中时，会向 GBP 注册该页。
- 尝试将某页访存到 LBP 中时，成员会先检查 GBP，然后仅当 GBP 中不存在此页时才从磁盘读取此页的内容(GBP 中存在的此页的版本决不会比磁盘上此页的版本旧)。
- 成员具有针对某页的更新锁定(X 方式下的 P-lock)时，成员的 LBP 中该页的版本可能比 GBP 中该页的版本新。
- 在成员释放更新锁定(或降低 P-lock 级别)之前，系统会使用该页的较新版本更新 GBP。
- 修改某页的事务结束(通过落实或回滚)时，已修改页会写至 GBP。
- 如果另一成员请求 P-lock 以读取某页的最新版本或更新该页，那么还可在事务结束之前通过页面协商将该页写至 GBP。另一成员请求 P-lock 时，此锁定冲突会导致拥有锁定的成员将已修改页写至 GBP 以便它可释放 P-lock 或使其降级，之后发出请求的成员可请求此锁定。

GBP 使用的总内存量由 cf_gbp_sz 数据库配置参数控制。第一次在成员上激活数据库时，如果 CF 上还没有 GBP，那么会分配 GBP。CF 停止时、数据库在整个集群中被删除或一致关闭时或在数据库恢复操作期间，此 GBP 被释放。

释放操作会将 GBP 中的页写至磁盘并在成员之间进行协调。释放操作类似于在 LBP 中清除页并实现两个功能：

- 将脏页写至磁盘以确保有足够的干净目录条目和数据元素可用于新页注册和写入。

- 通过确保 GBP 中没有页的保留时间超过指定时间来维护特定恢复窗口。这样做可以减少在恢复时要重演的日志记录数。

在 DB2 pureScale 集群环境中，SOFTMAX 等控制缓冲池刷新脏数据的方式已经失效。现在主要由 page_age_trgt_gcr 和 page_age_trgt_mcr 这两个数据库配置参数来控制释放行为。

通俗地说，GBP 提供了集群全局脏数据的缓冲池，所有的脏数据都会先存放到 GBP，再通过一定的规则从 GBP 持久化到磁盘。GBP 还要标记所有成员节点的本地缓冲池数据是否有效。成员节点事务访问数据时，首先检查本地缓冲池数据的有效性，然后从 GBP 获取最新的已经提交的数据，最后才从存储读取数据。

10.2.2　全局锁管理器

类似于组缓冲池与本地缓冲池的关系，DB2 pureScale 集群内部具有两种锁管理器：本地锁管理器(LLM)和全局锁管理器(GLM)。这两种锁管理器一起协作处理整个集群内部的并发控制。本地锁管理器在每个成员节点上，为当前成员的事务处理锁请求，它维护的是当前成员节点上的应用和事务；而全局锁管理器(GLM)是 CF 节点上的功能组件，处理的是所有成员节点上本地锁管理器(LLM)的申请，所以 GLM 维护了所有成员节点的锁请求。当一个事务申请锁时，这些锁是通过 LLM 和 GLM 协调而获取的。

LLM 和 GLM 的交互是通过 RDMA 协议完成的，并且都在内存里进行，所以沟通的成本并不算高。前面说到 P-lock 是 pureScale 环境中一种新的锁，而且还是所谓的物理锁。如果两个成员在修改同一个数据页的不同行数据，也会导致这个数据页物理锁的竞争。而这种竞争行为并不是一直对立，而是导致一种锁回收的行为。

假设成员 1 的事务 A 拥有这个 P-lock，修改了一行数据，成员 2 的事务 B 想要申请同样的 P-lock，写另外一条数据。事务 B 并不需要一直等到事务 A 提交并释放这个锁，而是通过和 CF 交互，让事务 A 写完日志，并且等待成员 1 将数据页的脏数据写入 CF 的 GBP，那么这时事务 B 就可以获取这个 P-lock 并从 GBP 获取最新版本的数据页以进行下一步操作。这种行为称为页回收，也被称为锁回收。在监控和调优 DB2 集群的过程中，这种行为的代价很高，需要特别关注并调优。

10.2.3　DB2 pureScale 集群相关参数

这里只列出 pureScale 相关参数以及对 pureScale 性能有影响的参数。

1. 注册表变量

- DB2_ALLOW_WLB_WITH_SEQUENCES 注册表变量控制是否允许用来访问序列的应用程序参与工作负载均衡。当 DB2_ALLOW_WLB_WITH_SEQUENCES 设置为 NO 时，会阻止在 SQL 序列语句中引用 PREVIOUS VALUE 或 NEXT VALUE 的应用程序参与工作负载均衡。默认是 NO，建议设置为 YES。
- DB2_DATABASE_CF_MEMORY 用于指示将 cf_db_mem_sz 数据库配置参数设置为 AUTOMATIC 的每个数据库的总 CF 内存(CF_MEM_SZ)比例。cf_db_mem_sz 被设置为特定值的任何数据库将忽略此注册表变量。这个变量初始建议设置为-1，那么数据库在 cf_db_mem_sz 设置为自动的情况下，自动获取的大小为 CF_MEM_SZ/NUMDB，而 cf_db_mem_sz 建议设置为固定值，以覆盖 DB2_DATABASE_CF_MEMORY 设置。
- DB2_MCR_RECOVERY_PARALLELISM_CAP：在多数据库环境中，如果需要成员崩溃恢复(MCR)，那么在每个成员上并行恢复的数据库数由 numdb 配置参数或 DB2_MCR_RECOVERY_PARALLELISM_CAP 注册表变量的值(两者中的较小者)设置。
- DB2_SD_ALLOW_SLOW_NETWORK 变量允许小于 10GE 的以太网卡使用 cf_transport_method 配置参数的 TCP 选项。这种具有较低传输速率的以太网卡会限制性能。一般只有在测试环境或开启了 EHL 的 pureScale 集群生产环境中才使用小于 10GE 的以太网卡。

2. 实例变量

- CF_MEM_SZ 参数控制集群高速缓存设施(又称为 CF)使用的总内存。如果应用默认设置为 AUTOMATIC，那么集群高速缓存设施的内存量通过查询 CF 服务器上可用的总内存确定。然后此参数被设为 CF 服务器上总内存的适当百分比或机器上的可用内存量(取两者中的较小值)。适当百分比通常为 CF 上可用总内存的 70%到 90%。建议设置为固定值。
- cf_num_conns 参数控制集群高速缓存设施(CF)连接池的初始大小。如果将 cf_num_conns 参数设为固定数字值，那么对于每个 CF，DB2 数据库管理器会在每个成员启动时为该成员创建正好等于该数目的 CF 连接。DB2 数据库管理器不会进行任何自动增长或缩减。建议保持默认值 AUTOMATIC。
- cf_num_workers 参数指定集群高速缓存设施(CF)上的工作程序线程总数。工作程序线程分布在多个通信适配器端口之间以平衡每个接口上用于处理请求的工作程序线程数。如果设为默认值(AUTOMATIC)，那么参数值配置为 CF 上的可用处理

器数减 1。可用 CPU 在实例间平均划分，然后从针对每个实例生成的值中减一个处理器。如果 CF 主机上有共存成员，那么 CF 上的工作程序线程数进一步除以主机上的 CF 和成员的总数。如果 CF 服务器上只有一个处理器，那么此值设为 1。CF 工作程序线程总数显示在 cfdiag.log 文件中。如果手动设置 cf_num_workers 参数，请将该值设置为等于或大于通信适配器端口数以便每个接口至少有一个工作程序线程。如果工作程序线程数不足以覆盖所有接口，那么系统会针对启动失败的 CF 记录警报。必须更改该参数值才能解决此问题。在使用 TCP/IP 作为通信方式的 DB2 pureScale 环境中，当 cf_num_workers 设置为默认值(AUTOMATIC)时，会将参数值配置为实例上可用处理器数的四倍。如果正运行 InfiniBand 或 uDAPL，请不要将此值设为大于 CF 服务器上的处理器数。每个工作程序线程在一个处理器上运行以等待 uDAPL 通信。如果工作程序线程数超过处理器数，那么性能会受影响。如果 CF 节点有其他服务，就设定为固定值，值的大小为节点 CPU 数减去想要保留的 CPU 数。

- cf_transport_method 配置参数控制将哪种方法用于在 DB2 成员与集群高速缓存工具(CF)之间进行通信。当 cf_transport_method 设置为 RDMA 时，DB2 成员必须通过使用 Remote Direct Memory Access (RDMA)来与 CF 通信。要使用 RDMA 协议网络，必须使用的硬件配置是安装了相应 InfiniBand 硬件的 InfiniBand 网络或是使用 RoCE 适配卡的以太网网络。当 cf_transport_method 参数设置为 TCP 时，DB2 成员通过使用 TCP/IP 协议网络来与 CF 通信。
- cf_diagpath 指定 CF 的诊断信息文件的标准路径，默认是在“INSTHOME/sqllib/db2dump/ $m”下面。建议修改，不要放在 GPFS 文件系统和实例目录下。同样，对 diagpath 也建议如此。
- cf_diaglevel 参数指定将记录在 cfdiag*.log 文件中的诊断错误类型，默认为 2，记录所有错误。不建议修改。

3. 数据库变量

- cf_db_mem_sz 参数控制此数据库的集群高速缓存设施(又称为 CF)的总内存限制。cf_gbp_sz、cf_lock_sz 和 cf_sca_sz 参数的集群高速缓存设施(CF)结构内存限制之和必须小于 cf_db_mem_sz 参数的 CF 结构内存限制。在 cf_db_mem_sz 参数绑定的同一数据库内的 CF 资源(例如组缓冲池(GBP)、共享通信区(SCA)和锁定结构)之间自动调整内存。如果设置为固定值，需要知道有额外 3840 个 4KB 页(取自 cf_mem_sz 参数)将由 CF 内部使用。此外，cf_mem_sz 参数中会额外保留 256 个 4KB 页以供实例中的每个活动数据库使用。建议设定为固定值。

- cf_gbp_sz 参数确定集群高速缓存设施(又称为 CF)用于此数据库的组缓冲池(GBP)的内存大小。建议设置为固定值，大小为所有成员节点 LBP 之和的 30%～40%。
- cf_lock_sz 参数确定 CF 用于此数据库的锁定的内存大小。建议设置为固定值。可使用 current_cf_lock_size 监视元素来监视 CF 锁定内存的消耗。
- cf_sca_sz 参数确定集群高速缓存设施(CF)中 SCA 使用的内存大小。该 SCA 对应每个数据库实体，并且包含表、索引、表空间和目录的数据库范围控制块信息。此参数可联机配置，建议使用默认值 AUTOMATIC。DB2 数据库管理器会在对任何成员第一次激活数据库时计算内存值，此值应足以供基本数据库操作使用。
- page_age_trgt_mcr 参数指定在将已更改的页面持久存储到表空间存储器(或者对于 DB2 pureScale 实例，持久存储到表空间存储器或组缓冲池)之前要在本地缓冲池中保留这些页面的目标持续时间(以秒计)。当 softmax 参数配置为值 0 时，会使用 page_age_trgt_mcr 参数。已迁移的数据库会保留 softmax 的上一个值，并且忽略 page_age_trgt_mcr 参数(如果此值不是 0)。在新数据库中，softmax 的值被设置为 0。当 softmax 参数设置为 0 时，page_age_trgt_mcr 参数用于确定软检查点的频率。增大此配置参数的值会将已更改的页面在内存中保留更久，从而允许在将这些页面持久存储到磁盘之前缓冲更多页面更新。此行为可帮助提高性能，但是也会延长恢复时间。
- page_age_trgt_gcr 参数指定在将已更改的页面持久存储到磁盘或高速缓存设施之前，要在组缓冲池中保留这些页面的目标持续时间(以秒计)。page_age_trgt_gcr 参数必须大于或等于 page_age_trgt_mcr 数据库配置参数。当 softmax 参数配置为值 0 时，会使用 page_age_trgt_gcr 参数。已迁移的数据库会保留 softmax 的上一个值，并且忽略 page_age_trgt_gcr 参数(如果此值不是 0)。在新数据库中，softmax 的值被设置为 0。增大此配置参数的值会将已更改的页面在内存中保留更久，从而允许在将这些页面持久存储到磁盘之前缓冲更多页面更新。此行为可帮助提高性能，但是也会延长恢复时间。
- softmax 参数被替换为新的 page_age_trgt_mcr 和 page_age_trgt_gcr 参数(这两个参数都配置为几秒)，所以在 pureScale 环境中这个值为 0。
- opt_direct_wrkld 参数启用或禁用显式分层锁定(EHL)。opt_direct_wrkld 参数指定表或范围分区是否有资格进入 NOT_SHARED 状态。它全局适用于所有 DB2 pureScale 成员。进入 NOT_SHARED 状态的表，会保持处于此状态，直到另一成员访问该表。在实际运用中表状态的改变过程存在性能下降问题，所以不建议在多节点访问的生产环境中使用。除非仅仅把 pureScale 当作高可用集群，应用指

定节点访问，其他节点为热备状态的特殊场景。这种情况下设置此参数为 ON，能减少与 CF 节点的交互，获取更高的性能。

10.3　DB2 集群性能监控

DB2 集群的所有事务处理都和单机版不一样，无论是数据的访问还是锁的申请，都需要和 CF 交互。所以在 DB2 集群调优过程中，最需要关注的就是这些不一样的地方。DB2 集群加入了很多新的监控指标和监控工具，下面就一步步看如何监控 DB2 集群的性能并加以优化。

10.3.1　查看 CF 资源利用

CF 节点的系统资源已经预先分配，cf_num_workers 参数控制系统 CPU 资源占用，CF_MEM_SZ 控制系统内存占用。

因为 DB2 集群的 CF 节点会一直占用操作系统 CPU，而在 DB2 集群内部，这些 CPU 资源是否充足，可以通过查询 SYSIBMADM.ENV_CF_SYS_RESOURCES 来获取实际用量。

```
SELECT VARCHAR(NAME,20) AS ATTRIBUTE,VARCHAR(VALUE,25) AS VALUE,
VARCHAR(UNIT,8) AS UNIT FROM SYSIBMADM.ENV_CF_SYS_RESOURCES
```

输出结果如图 10-3 所示：

```
ATTRIBUTE            VALUE                     UNIT
-------------------- ------------------------- --------
HOST_NAME            AGDPCCF1                  -
MEMORY_TOTAL         30720                     MB
MEMORY_FREE          18804                     MB
MEMORY_SWAP_TOTAL    16384                     MB
MEMORY_SWAP_FREE     16351                     MB
VIRTUAL_MEM_TOTAL    47104                     MB
VIRTUAL_MEM_FREE     35156                     MB
CPU_USAGE_TOTAL      0                         PERCENT
HOST_NAME            BGDPCCF1                  -
MEMORY_TOTAL         30720                     MB
MEMORY_FREE          19502                     MB
MEMORY_SWAP_TOTAL    16384                     MB
MEMORY_SWAP_FREE     16352                     MB
VIRTUAL_MEM_TOTAL    47104                     MB
VIRTUAL_MEM_FREE     35855                     MB
CPU_USAGE_TOTAL      0                         PERCENT
```

图 10-3　CF 资源利用

这个管理视图显示了 CF 节点上的内存和实际 CF 占用的 CPU 量。其中，CPU 实际使用量和在操作系统里面看到的是不一样的。一般 CPU_USAGE_TOTAL 在主 CF 节点的值会略大于在备 CF 节点的值。主 CF 节点的这个值也不会很高，一般在 10%以内，繁忙的时候也不会超过 30%。但是如果真的遇到性能问题，尤其是和 CF 交互相关的性能问题，这个值可能瞬间会比较高。这种情况下就需要分析当前的事务性能。CF 交互主要就是读写数据页、申请和释放锁等。当出现热点数据或者进行大量数据处理的时候，这个值可能会变高。

10.3.2 查看各个成员的负载情况

DB2 集群内多个成员节点同时提供数据库服务。应用服务器可以通过开启工作负载均衡(WLM)的方式由 DB2 集群协调负载分发，充分利用系统资源。当然也可以通过指定访问的成员，或是完全轮询访问成员节点来分发负载。无论使用哪一种方式，数据库管理员都希望能清楚知晓 DB2 集群内部的工作负载实际状态。db2pd 工具提供了相关监指标，查看 serverlist，输出结果如图 10-4 所示：

```
AGDPCMB1:/home/db2gdpc$db2pd -d chgmdb -serverlist

Database Member 0 -- Active -- Up 0 days 02:42:17 -- Date 2017-04-12-14.16.41.803102

Server List:
   Time:                Wed Apr 12 11:38:45
   Database Name:       CHGMDB
   Member Subset Id:    0
   Member Subset Name:  SYSDEFAULTMEMBERSUBSET
   Count:               4
   Member     Hostname                          Non-SSL Port    SSL Port    Priority
   0          AGDPCMB1                          50000           0           61
   3          BGDPCMB2                          50000           0           100
   1          AGDPCMB2                          50000           0           77
   2          BGDPCMB1                          50000           0           86
```

图 10-4　集群负载情况

在这个输出结果中主要关注 Priority 值，越大说明越空闲，越小说明越繁忙。如果是 0，通常表示对应的节点不可用。这个值是瞬间值，通常都不会很平均，需要持续观察才能确定负载分布是否均衡。

10.3.3 查看缓冲池命中率

DB2 集群内部有全局缓冲池和本地缓冲池，监控指标也多了很多项。通常在单机版数据库环境中，需要通过计算缓冲池的命中率来分析是否存在性能问题，无论是缓冲池设置过小，还是发生了昂贵的大查询。这些是计算缓冲池命中率的意义，因为在缓冲池找到数据比从磁盘读取到数据的成本低很多。在 DB2 集群环境内，应用访问需要先访问 LBP 来尝试获取数据，如果没有找到，那么再去 GBP 尝试获取数据。如果都没有，最后才是从磁

盘获取。访问 GBP 是通过 RDMA 协议从 CF 节点远程获取，时间大约是 30 纳秒(如果不是同城双活 GDPC 环境)。这个速度相对于从磁盘获取数据是相当快的，几乎约等于从 LBP 获取到了数据。所以要计算缓冲池的命中率来判断是否存在物理读太多的问题，应该计算从 LBP 加上 GBP 获取到的次数后除以全部获取次数。

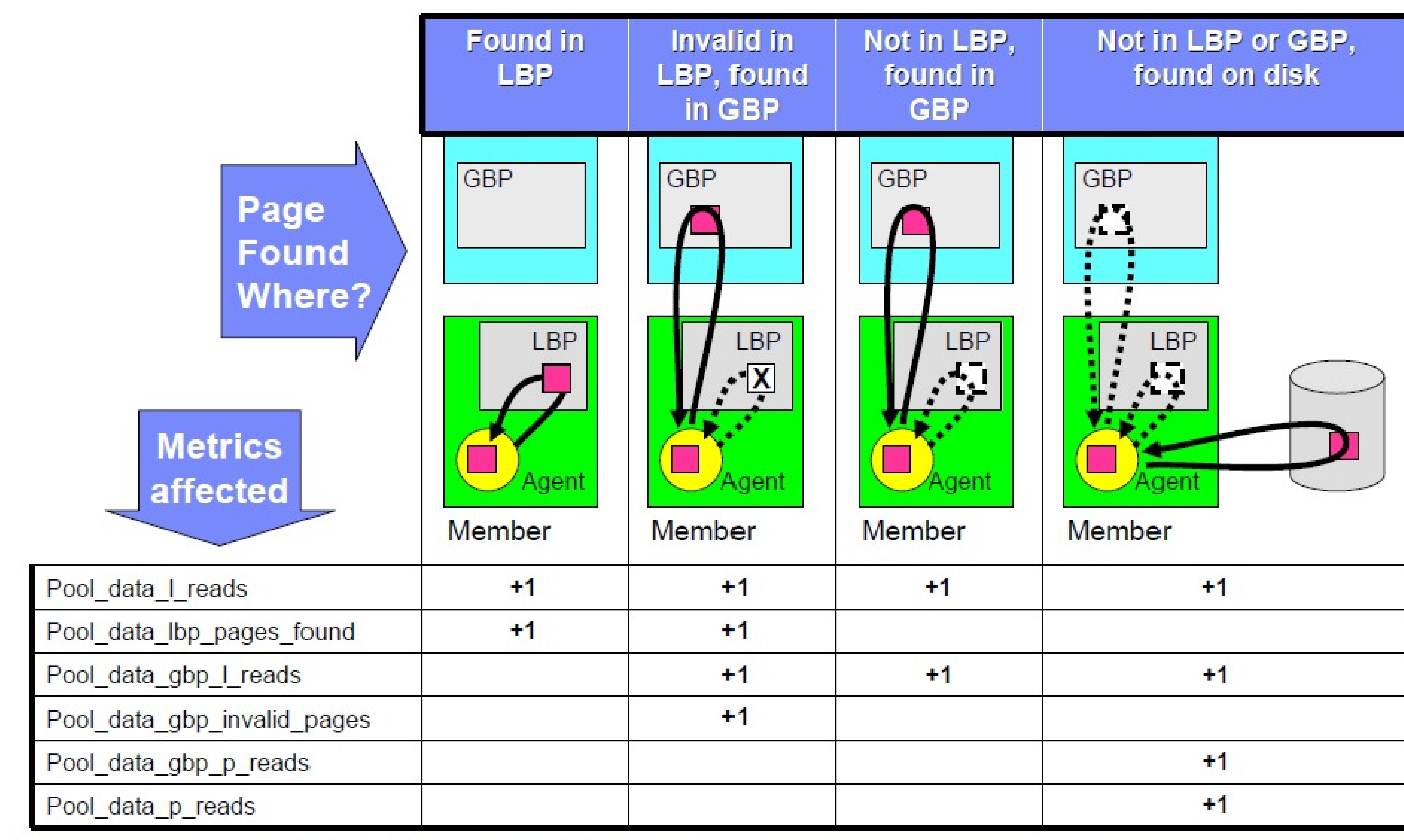
在此之前最先要了解的是监控指标的含义。下面列出缓冲池相关的监控指标：

- pool_data_l_reads 监视元素指示从常规表空间和大型表空间的逻辑缓冲池中请求获取的数据页的数目。
- pool_data_p_reads 监视元素指示从常规表空间和大型表空间的物理表空间容器中读取的数据页数。
- pool_data_lbp_pages_found 监视元素指示数据页出现在本地缓冲池中的次数。
- pool_data_gbp_indep_pages_found_in_lbp 监视元素指示代理程序在本地缓冲池(LBP)中发现的独立于组缓冲池(GBP)的数据页数。在 DB2 pureScale 环境外部，此值为空。
- pool_data_gbp_invalid_pages 监视元素指示数据页在本地缓冲池中无效并因此改为从组缓冲池中读取的次数。在 DB2 pureScale 环境外部，此值为空。
- pool_data_gbp_l_reads 监视元素指示尝试从组缓冲池读取依赖于组缓冲池(GBP)的数据页(因为该页在本地缓冲池(LBP)中无效或不存在)的次数。在 DB2 pureScale 环境外部，此值为空。
- pool_data_gbp_p_reads 监视元素指示尝试将磁盘中依赖于组缓冲池(GBP)的数据页读取到本地缓冲池中(因为在 GBP 中找不到该页)的次数。在 DB2 pureScale 环境外部，此值为空。
- pool_async_data_reads 监视元素指示异步引擎可派遣单元(EDU)从所有类型的表空间的物理表空间容器中读取的数据页数。
- pool_async_data_lbp_pages_found 监视元素指示预取程序尝试访问数据页时此数据页出现在本地缓冲池中的次数。
- pool_async_data_gbp_indep_pages_found_in_lbp 监视元素指示本地缓冲池中由异步 EDU 发现的独立于组缓冲池(GBP)的数据页数。
- pool_async_data_gbp_invalid_pages 监视元素指示预取程序尝试从组缓冲池读取数据页(因为该数据页在本地缓冲池中无效)的次数。在 DB2 pureScale 环境外部，此值为空。
- pool_async_data_gbp_l_reads 监视元素指示预取程序尝试从组缓冲池读取依赖于组缓冲池(GBP)的数据页(因为该页在本地缓冲池中无效或不存在)的次数。在 DB2 pureScale 环境外部，此值为空。

- pool_async_data_gbp_p_reads 监视元素指示预取程序尝试将磁盘中依赖于组缓冲池(GBP)的数据页读取到本地缓冲池中(因为在 GBP 中找不到该数据页)的次数。在 DB2 pureScale 环境外部，此值为空。
- pool_index_l_reads 监视元素指示从常规表空间和大型表空间的逻辑缓冲池中请求获取的索引页数。
- pool_index_p_reads 监视元素指示从常规表空间和大型表空间的物理表空间容器中读取的索引页数。
- pool_index_lbp_pages_found 监视元素指示索引页出现在本地缓冲池中的次数。
- pool_index_gbp_indep_pages_found_in_lbp 监视元素指示代理程序在本地缓冲池(LBP)中发现的独立于组缓冲池(GBP)的索引页数。在 DB2 pureScale 环境外部，此值为空。
- pool_index_gbp_invalid_pages 监视元素指示尝试从组缓冲池读取索引页(因为该数据页在本地缓冲池中无效)的次数。在 DB2 pureScale 环境外部，此值为空。
- pool_index_gbp_l_reads 监视元素指示尝试从组缓冲池读取依赖于组缓冲池(GBP)的索引页(因为该数据页在本地缓冲池中无效或不存在)的次数。在 DB2 pureScale 环境外部，此值为空。
- pool_index_gbp_p_reads 监视元素指示尝试将磁盘中依赖于组缓冲池(GBP)的索引页读取到本地缓冲池中(因为在 GBP 中找不到该数据页)的次数。在 DB2 pureScale 环境外部，此值为空。
- pool_async_index_reads 监视元素指示异步引擎可派遣单元(EDU)从所有类型的表空间的物理表空间容器中读取的索引页数。
- pool_async_index_lbp_pages_found 监视元素指示预取程序尝试访问索引页时此索引页出现在本地缓冲池中的次数。
- pool_async_index_gbp_indep_pages_found_in_lbp 监视元素指示本地缓冲池中由异步 EDU 发现的独立于组缓冲池(GBP)的索引页数。
- pool_async_index_gbp_invalid_pages 监视元素指示预取程序尝试从组缓冲池读取索引页(因为该数据页在本地缓冲池中无效)的次数。在 DB2 pureScale 环境外部，此值为空。
- pool_async_index_gbp_l_reads 监视元素指示预取程序尝试从组缓冲池读取依赖于组缓冲池(GBP)的索引页(因为该数据页在本地缓冲池中无效或不存在)的次数。在 DB2 pureScale 环境外部，此值为空。

- pool_async_index_gbp_p_reads 监视元素指示预取程序尝试将磁盘中依赖于组缓冲池(GBP)的索引页面读取到本地缓冲池中(因为在 GBP 中找不到该数据页)的次数。在 DB2 pureScale 环境外部，此值为空。
- pool_temp_data_l_reads 监视元素指示向临时表空间的逻辑缓冲池请求的数据页数。
- pool_temp_data_p_reads 监视元素指示从临时表空间的物理表空间容器中读取的数据页数。
- pool_temp_index_l_reads 监视元素指示向临时表空间的逻辑缓冲池请求的索引页数。
- pool_temp_index_p_reads 监视元素指示从临时表空间的物理表空间容器中读取的索引页数。

下面通过图 10-5 来了解上述参数的意义，以 data 为例：



	Found in LBP	Invalid in LBP, found in GBP	Not in LBP, found in GBP	Not in LBP or GBP, found on disk
Pool_data_l_reads	+1	+1	+1	+1
Pool_data_lbp_pages_found	+1	+1		
Pool_data_gbp_l_reads		+1	+1	+1
Pool_data_gbp_invalid_pages		+1		
Pool_data_gbp_p_reads				+1
Pool_data_p_reads				+1

图 10-5　监控元素示例

图 10-5 中显示了获取数据的流程和相应的监控元素计数器变化。其中本地缓冲池找到对应的数据页元素 pool_data_lbp_pages_found 包含了本地缓冲池找到的有效和无效数据页，也就是包含 pool_data_gbp_invalid_pages 监视元素；而 pool_data_gbp_p_reads 和 pool_data_p_reads 监视元素应该是几乎一致的，都是从本地缓冲池 LBP 和组缓冲池 GBP 没有找到数据页而发生了物理读。获取到数据页的成本是 LBP<GBP<磁盘。上面这个示意

图并没有区分同步读和异步读，而是包含了这两种情况。异步读是 DB2 引擎帮助预先读取需要的数据页到缓冲池内，这样应用实际访问的时候从缓冲池获取的概率更高，提高了命中率。而应用访问数据的缓冲池命中率是最应该关心的。所以计算缓冲池命中率的时候，建议去掉异步读产生的偏差。也就是只要不发生同步读，就相当于在缓冲池命中了。但是异步读里面包含了常规表空间和临时表空间的数据读。

这些监控元素可以从不同的维度收集：表空间、缓冲池和数据库。建议使用 DB2 提供的表函数来收集。这些表函数包括 MON_GET_BUFFERPOOL、MON_GET_DATABASE、MON_GET_DATABASE_DETAILS 和 MON_GET_TABLESPACE。

1. 数据同步读公式

数据的同步读要去掉异步读计数器：

```
    数据同步读数量 = pool_data_p_reads + pool_temp_data_p_reads -
pool_async_data_reads
```

所以这里建议临时表空间使用单独的缓冲池，这样就能确切知道数据请求的缓冲池命中率是什么了，而数据表空间对应的缓冲池 pool_temp_data_p_reads 监控元素将是 0。但是如果临时表空间使用了同样的缓冲池，在计算命中率的时候还是要考虑临时表空间访问的影响。

2. 整体缓冲池命中率公式

以数据页的访问为例，数据整体缓冲池的命中率如下所示：

```
    整体缓冲池命中率=(pool_data_l_reads + pool_temp_data_l_reads -
(pool_data_p_reads - pool_async_data_reads)) / (pool_data_l_reads  +
pool_temp_data_l_reads)
```

下面这个例子是通过 MON_GET_BUFFERPOOL 表函数检查所有缓冲池在所有节点的数据页整体命中率：

```
    ~> db2 "SELECT MEMBER, VARCHAR(bp_name, 20)AS bp_name,
    > DEC((float(pool_data_l_reads + pool_temp_data_l_reads -
(pool_data_p_reads - pool_async_data_reads))/ float(pool_data_l_reads +
pool_temp_data_l_reads))* 100, 5, 2) DATA_OVERALL_HIT
    > FROM TABLE (MON_GET_BUFFERPOOL('', -2)) where bp_name not like 'IBMSYS%'
and pool_data_l_reads > 0  order by member,bp_name"
```

```
MEMBER BP_NAME              DATA_OVERALL_HIT
------ -------------------- ----------------
     0 IBMDEFAULTBP                    99.81
     1 IBMDEFAULTBP                    97.57

  2 record(s) selected.
```

索引的整体缓冲池命中率与此相似，替换公式中的 data 为 index 即可：

```
    数据整体缓冲池命中率=(pool_index_l_reads + pool_temp_index_l_reads -
(pool index p reads - pool async index reads)) / (pool index l reads  +
pool_temp_index_l_reads)
```

同样通过 MON_GET_BUFFERPOOL 表函数检查所有缓冲池在所有节点的索引页的整体命中率：

```
    ~> db2 "SELECT MEMBER, VARCHAR(bp_name, 20)AS bp_name,
    > DEC((float(pool_index_l_reads + pool_temp_index_l_reads -
(pool_index_p_reads - pool_async_index_reads))/ float(pool_index_l_reads +
pool temp index l reads))* 100, 5, 2) IDX OVERALL HIT
    > FROM TABLE (MON_GET_BUFFERPOOL('', -2)) where bp_name not like 'IBMSYS%'
and pool_index_l_reads > 0  order by member,bp_name"

    MEMBER BP_NAME              IDX_OVERALL_HIT
    ------ -------------------- ---------------
         0 IBMDEFAULTBP                   99.91
         1 IBMDEFAULTBP                   56.77

      2 record(s) selected.
```

对于使用比较频繁的缓冲池，数据的缓冲池命中率期望大于 90%，索引的缓冲池命中率期望大于 95%，否则需要调查缓冲池的设置是否有效。这个命中率是从整体来看的，影响的因素很多，还需要进一步分析是本地缓冲池 LBP 设置过小，还是组缓冲池 GBP 效率比较差，又或者是不同类型的负载冲突。

3. 本地缓冲池命中率公式

以数据页的访问为例，数据本地缓冲池的命中率如下所示：

```
    数据本地缓冲池的命中率=(pool_data_lbp_pages_found -
pool_async_data_lbp_pages_found) / (pool_data_l_reads +
pool_temp_data_l_reads )
```

下面这个例子是通过 MON_GET_BUFFERPOOL 表函数检查所有缓冲池在所有节点的数据页本地命中率：

```
    ~> db2 "SELECT MEMBER, VARCHAR(bp_name, 20)AS bp_name,
    > DEC((float(pool_data_lbp_pages_found  -
pool_async_data_lbp_pages_found )/ float(pool_data_l_reads +
pool_temp_data_l_reads))* 100, 5, 2) DATA_LBP_HIT
    > FROM TABLE (MON_GET_BUFFERPOOL('', -2)) where bp_name not like 'IBMSYS%'
and pool_data_l_reads > 0 order by member,bp_name"

    MEMBER BP_NAME               DATA_LBP_HIT
    ------ -------------------- ------------
         0 IBMDEFAULTBP                99.55
         1 IBMDEFAULTBP                97.54

      2 record(s) selected.
```

相对于整体缓冲池命中率，本地缓冲池的命中率会低一些，因为去除了从 GBP 获得数据的计数。如果本地缓冲池命中率低，并不能说明 LBP 不够大，也有可能是因为访问的是热点数据。

索引的本地缓冲池命中率与此相似，替换公式中的 data 为 index 即可：

```
    索引本地缓冲池命中率=( pool_index_lbp_pages_found -
pool_async_index_lbp_pages_found) / (pool_index_l_reads +
pool_temp_index_l_reads)
```

同样通过 MON_GET_BUFFERPOOL 表函数检查所有缓冲池在所有节点的索引页本地命中率：

```
    ~> db2 "SELECT MEMBER, VARCHAR(bp_name, 20)AS bp_name,
    > DEC((float(pool_index_lbp_pages_found  -
pool_async_index_lbp_pages_found )/ float(pool_index_l_reads +
pool_temp_index_l_reads))* 100, 5, 2) INDEX_LBP_HIT
```

```
    > FROM TABLE (MON_GET_BUFFERPOOL('', -2)) where bp_name not like 'IBMSYS%'
and pool_data_l_reads > 0 order by member,bp_name"

    MEMBER BP_NAME              INDEX_LBP_HIT
    ------ -------------------- -------------
         0 IBMDEFAULTBP                 99.94
         1 IBMDEFAULTBP                 74.29

      2 record(s) selected.
```

4. 组缓冲池命中率公式

以数据页的访问为例，数据组缓冲池的命中率如下所示：

```
    数据组缓冲池的命中率=((pool_data_gbp_l_reads - pool_data_gbp_p_reads) -
(pool_async_data_gbp_l_reads - pool_async_data_gbp_p_reads )) /
(pool_data_l_reads + pool_temp_data_l_reads )
```

下面这个例子是通过 MON_GET_BUFFERPOOL 表函数检查所有缓冲池在所有节点的数据组缓冲池的命中率：

```
     ~> db2 "SELECT MEMBER, VARCHAR(bp_name, 20)AS bp_name,
    > DEC((float((pool data gbp l reads - pool data gbp p reads) -
(pool_async_data_gbp_l_reads - pool_async_data_gbp_p_reads ))/
float(pool_data_l_reads + pool_temp_data_l_reads))* 100, 5, 2) DATA_GBP_HIT
    > FROM TABLE (MON_GET_BUFFERPOOL('', -2)) where bp_name not like 'IBMSYS%'
and pool_data_l_reads > 0 order by member,bp_name"

    MEMBER BP_NAME              DATA_GBP_HIT
    ------ -------------------- ------------
         0 IBMDEFAULTBP                 0.00
         1 IBMDEFAULTBP                 0.09

      2 record(s) selected.
```

索引组缓冲池的命中率与此相似，替换公式中的 data 为 index 即可：

```
    索引组缓冲池的命中率=((pool_index_gbp_l_reads - pool_index_gbp_p_reads) -
(pool_async_index_gbp_l_reads - pool_async_index_gbp_p_reads )) /
```

```
(pool_index_l_reads + pool_temp_index_l_reads )
```

同样通过 MON_GET_BUFFERPOOL 表函数检查所有缓冲池在所有节点的索引页组缓冲池的命中率：

```
    ~> db2 "SELECT MEMBER, VARCHAR(bp_name, 20)AS bp_name,
    > DEC((float((pool_index_gbp_l_reads - pool_index_gbp_p_reads) -
(pool_async_index_gbp_l_reads - pool_async_index_gbp_p_reads ))/
float(pool_index_l_reads+ pool_temp_index_l_reads))* 100, 5, 2) INDEX_GBP_HIT
    > FROM TABLE (MON GET BUFFERPOOL('', -2)) where bp name not like 'IBMSYS%'
and pool_data_l_reads > 0 order by member,bp_name"

    MEMBER BP_NAME              INDEX_GBP_HIT
    ------ -------------------- -------------
         0 IBMDEFAULTBP                  0.00
         1 IBMDEFAULTBP                  0.10

      2 record(s) selected.
```

组缓冲池的命中率通常会很低。这并不会说明性能很差而需要扩大组缓冲池。因为影响事务访问数据的主要参考是全局的缓冲池命中率。如果想要知道 GBP 的大小设置是否足够，可以监控 GBP 满的情况。如果 GBP 设置过小，经常出现 GBP 满，会导致事务更新数据的时候，需要等待 GBP 写完数据页才有空间。通过 MON_GET_BUFFERPOOL 表函数检查 num_gbp_full 的次数，并且和 commit_sql_stmts 参数比较：

```
    ~> db2 "select 10000.0 * sum(mggb.num gbp full) / sum(commit sql stmts) from
table(mon_get_group_bufferpool(-2)) as mggb, sysibmadm.snapdb"

    1
    ---------------------------------
                        0.0000000

      1 record(s) selected.
```

这个比例如果比较高，那么真的需要扩大 GBP 的设置。

10.3.4 查看全局锁性能

DB2 集群内部有全局锁，也就是锁的竞争发生在不同成员之间。因此需要监控与全局

锁管理相关的锁等待、锁超时、锁升级等相关性能指标。DB2 提供了多个表函数，可以列出全局锁管理的这些监控元素。这里使用其中一个表函数 MON_GET_PKG_CACHE_STMT 来示范。这个表函数会收集当前数据库的 PACKAGE CASH 里面的信息，返回一张表，可以被 SQL 语句直接调用。

```
    ~> db2 "SELECT MEMBER, LOCK_WAITS_GLOBAL,LOCK_WAIT_TIME_GLOBAL,
    >           LOCK_TIMEOUTS_GLOBAL,LOCK_ESCALS_GLOBAL,
    >           SUBSTR(stmt_text,1,50) as stmt_text
    >     FROM TABLE(MON_GET_PKG_CACHE_STMT ( 'D', NULL, NULL, -2)) as T
    >     WHERE LOCK WAITS GLOBAL <> 0
    >     order by LOCK_WAIT_TIME_GLOBAL desc"

    MEMBER LOCK_WAITS_GLOBAL    LOCK_WAIT_TIME_GLOBAL LOCK_TIMEOUTS_GLOBAL
LOCK_ESCALS_GLOBAL   STMT_TEXT
    ------ -------------------- -------------------- --------------------


        1                 1                811                 0
0 select * from t_sett_stat
        1                 2                379                 0
0 select * from t_product_info
        1                 1                343                 0
0 select * from t_sett_stat where mer_no like '0002%
        0                 1                298                 0
0 select *  from t_task  where  bat_id = ?  and  job
        0                 1                 68                 0
0 SELECT * FROM t_product_info WHERE prd_id =?
        0                 3                 45                 0
0 SELECT tx_stat, ret_msg, ext_data, tx_code FROM v_
        1                 2                 39                 0
0 select * from v_jrnl where chnl_seq_no like'816012
        1                 1                 28                 0
0 select * from t product info where prd id='0360018
        1                 1                 27                 0
0 select * from t_chnl_bank
        1                 1                 27                 0
0 select * from t_merchant_acct where mer_no like '9
        1                 1                 26                 0
```

```
0 select * from t_merchant_acct where mer_no like '0
          1                    1                    26                    0
0 select * from t_merchant_acct where contract_ID='8
          0                    1                    13                    0
0 SELECT * FROM t_ctrl_bw WHERE contract_id =? AND s
          0                    1                    12                    0
0 select * from t_busi_acct_lmt_def where (mer_no =?
          0                    1                    11                    0
0 SELECT para_val FROM t_pu_para WHERE para_name ='U
          0                    1                    11                    0
0 SELECT COUNT(*) FROM t_task_inst WHERE (task_stat
          0                    1                    11                    0
0 SELECT distinct mer_no,prd_id,contract_id,curr_cod
          0                    1                    11                    0
0 SELECT * FROM t_merchant_acct WHERE (mer_no,prd_id

     18 record(s) selected.
```

在调用 MON_GET_PKG_CACHE_STMT 返回的众多信息中，案例里面查询了 LOCK_WAITS_GLOBAL、LOCK_WAIT_TIME_GLOBAL、LOCK_TIMEOUTS_GLOBAL、LOCK_ESCALS_GLOBAL 等与全局锁管理相关的监控元素。

- LOCK_WAITS_GLOBAL：因为锁被其他 member 占用而等待的次数。次数越多，说明并发性受影响越大。
- LOCK_WAIT_TIME_GLOBAL：等待其他 member 占用的锁的时间。时间越多，说明并发性受影响越大。
- LOCK_TIMEOUTS_GLOBAL：因为等待其他 member 占用的锁而产生的超时次数。事务会因此失败而回滚。
- LOCK_ESCALS_GLOBAL：全局锁升级的次数。锁升级会导致并发性严重降低。

这个表函数对于在查询当前一段时间的系统性能问题是否是因为全局锁而引起非常有效。因为 Package Cache 的大小是有限的，所以早些时间运行的 SQL 语句可能已经被挤出这个内存块，就不会被这个表函数返回。DB2 还提供了其他一些针对不同对象的表函数，也能反馈这些全局锁的信息，例如 MON_GET_CONNECTION、MON_GET_WORKLOAD、MON_GET_UNIT_OF_WORK 等。

通过上面这个监控，可以了解到当前的热点语句和热点数据。全局锁是 member 之间的锁，也就是在不同的 member 上运行的应用需要访问同一张表的相同 page 内的数据。一

种解决方式就是将这些应用安排到同一个 member 上运行，就不存在 member 间竞争的问题。这种方式可以通过设置客户机偏好的方式进行。前提是定位到是哪些应用有冲突，单个 member 的系统资源是否足以运行这些应用。另一种方式是在应用的事务中加入更多的 commit。提交越频繁，锁释放得越快，从而降低锁冲突的概率。

如果观察到 LOCK_ESCALS_GLOBAL 出现，那么可能是因为 cf_lock_sz 数据库参数设置过小引起的，可以考虑增大 cf_lock_sz 数据库参数。

10.3.5　查看页回收(page Reclaiming)行为

DB2 集群的页回收行为是为了解决不同成员并发访问相同数据页的问题。虽然避免了完全串行，但是这种行为还是开销比较大，性能不理想。DB2 提供了 MON_GET_PAGE_ACCESS_INFO 表函数来提取相关的信息。这里需要关心的是什么表经常发生这样的行为，这种行为消耗了多少时间，这种行为发生在哪个数据库成员上。

```
~> db2 "SELECT SUBSTR(TABSCHEMA,1,8) AS TABSCHEMA,
         SUBSTR(TABNAME,1,20) AS TABNAME,
         RECLAIM_WAIT_TIME,
         MEMBER,
         SUBSTR(OBJTYPE,1,10) AS OBJTYPE
   FROM TABLE(MON_GET_PAGE_ACCESS_INFO(NULL,NULL,-2))
   WHERE RECLAIM_WAIT_TIME > 0
    ORDER BY RECLAIM_WAIT_TIME DESC
    FETCH FIRST 10 ROWS ONLY"

TABSCHEMA TABNAME              RECLAIM_WAIT_TIME    MEMBER  OBJTYPE
--------- -------------------- -------------------- ------ ----------
SYSIBM    SYSCOLUMNS                            162      0  TABLE
SYSIBM    SYSTABLES                              44      0  TABLE
SYSIBM    SYSCOLUMNS                             20      0  INDEX
SYSIBM    SYSTABLES                              11      0  INDEX
UNPSAPP   H_JRNL                                  6      0  INDEX
UNPSAPP   T_JRNL                                  1      0  INDEX
UNPSAPP   H_JRNL                                  1      0  INDEX
UNPSAPP   H_JRNL                                  1      0  INDEX
SYSIBM    SYSBUFFERPOOLS                          1      0  INDEX
SYSIBM    SYSPLAN                                 1      0  TABLE
```

```
    10 record(s) selected.
```

上面的这个方法可以找出因数据页回收而消耗时间最多的表。RECLAIM_WAIT_TIME 返回等待的时间，单位是毫秒。

Page Reclaiming 其实还是全局锁的问题。解决的思路和前述解决全局锁的问题差不多。对 Page Reclaiming 的监控能够更直观地发现时间消耗了多少，从而定位系统的性能问题在多大程度上与此相关。

10.4 DB2 集群设计调优

DB2 集群与单机版相比最大的开销就是不同成员之间的事务存在竞争。也就是说，热点数据的处理问题在 DB2 集群环境中放的很大。基本上 DB2 集群的调优手段都是围绕着如何打散热点数据页来进行的。

10.4.1 使用小的 pagesize

DB2 集群推荐使用 pagesize 小的表空间。尤其是热点表，如果查询、更新、插入操作比较多，就会在成员之间产生竞争。小一点的 pagesize 使一个页面放入的数据少，产生竞争的概率变小。

10.4.2 使用大的 extentsize

表空间使用了小的 pagesize，为了提高预取的效率，最好使用大的 extentsize。尤其是在有大量插入的场景中，能够观察到使用大的 extentsize 效果更好。

10.4.3 使用 LOB inline 方法

DB2 数据库提供了 LOB inline 方式来存放大对象。DB2 默认大对象是和行数据分开存放的。大对象的读取和写入也不走缓冲池，在数据库内部记录为 DirectRead 和 DirectWrite。但是有很多大对象数据其实并不是很大，完全可以放在一个页面内，和行数据放在一起。这样读写性能高，同时还能被压缩。LOB inline 方式就是为表定义一个 INLINE LENGTH，所有小于这个值的大对象都和数据存放在一起。

对于数据库集群，LOB inline 方式意味着更大的行记录，单个页面存放更少的行，减少了竞争概率。

10.4.4　使用大的 pctfree 设置

DB2 的表和索引都可以设置 pctfree，这个值是百分比，意思是在一个页面内部预留多少空间。这个参数在 load 和 reorg 的时候才生效，对正常的插入是没有作用的。所以通常情况下并不建议修改。同时设置更大的 pctfree，意味着需要更多的页来装数，也就是空间占用高。所以这个方法的适用场景要限制好：

- 小表：小表的空间占用少，即便设置大的 pctfree，也不会导致空间紧张。
- 热表：数据查询更新很多，并且集中，需要分散热点页。
- 插入量小：插入数据会塞满 pctfree 的空间，所以插入量大的表并不适合。

10.4.5　巧用 CURRENT MEMBER

DB2 在多节点环境中提供了 CURRENT MEMBER 变量，若应用连接在成员 0 节点，那么获取的这个 CURRENT MEMBER 变量就是 0。

```
AGDPCMB1:/chgm_db2arc/kzh$db2 "values current member"

1
-----------
          0

record(s) selected.
```

如果为表加入一列并插入 CURRENT MEMBER 的值，那么不同成员插入的值就不一样。首先，CURRENT MEMBER 的用处是解决热点索引页。

这里讨论两种不同类型的索引。一种索引是插入频繁的、顺序增长的数字，例如自增列、序列号、时间戳等。因为追加的行都落在索引的最后几页，会导致热点页问题。通过加入 CURRENT MEMBER 到索引里，每个成员插入的索引会在不同的页里。示例如下：

```
    alter table orders add column curmem smallint default current member
implicitly hidden;
    create index seqindex on ordernumber (curmem, seqnumber);
```

orders 表有个顺序增长的 seqnumber 列，加入 curmen 之后，不同成员插入的 curmem 不同，对应的索引页也不同，因此解决了热点索引页问题。

另一种索引类型是索引取值比较少，每次插入新的数据都是在原有的索引值上插入新的 RID，导致这些值对应的索引页成为热点。例如省份、邮编这样的列。这种情况下，可

以把 CURRENT MEMBER 加到索引的后面，也可以促使不同成员插入的数据落在不同的索引页面里。示例如下：

```
    alter table customer add column curmem smallint default current member
implicitly hidden;
    create index stateidx on customer (state, curmem);
```

10.4.6 巧用随机索引

随机索引是 DB2 为了解决热点页问题提供的新功能。DB2 的索引一般都是升序或降序排列。DB2 从 V10 开始提供了随机索引。简单来说，随机索引就是将索引的列值按照哈希算法分裂，然后对同样的哈希值页面进行再排序。所以随机索引对于按照值定向查询是很快的，但是对于范围取值的效率会低于常规索引。

示例如下：

```
    CREATE TABLE "DB2PURE"."SMS_CODE"  (
                   "SC_TASKID" VARCHAR(32 OCTETS) NOT NULL ,
                   "SC_MOBILE" VARCHAR(20 OCTETS) ,
                   "SC_CODE" VARCHAR(40 OCTETS) ,
                   "SC_COUNT" INTEGER ,
                   "SC CREATETIME" TIMESTAMP ,
                   "SC_EXPIREDTIME" TIMESTAMP)
                  ORGANIZE BY ROW;

    CREATE UNIQUE INDEX "DB2PURE"." SC_CREATETIME" ON "DB2PURE"."SMS _CODE"
                  ("SC_CREATETIME" RANDOM)
                  INCLUDE NULL KEYS ALLOW REVERSE SCANS;
```

上面介绍了在 DB2 集群里一些常用的调优方法，中心思想就是通过数据库内部的技术手段打散热点数据页，减少成员之间的竞争。事实上从上层应用角度考虑，通过区分不同的工作负载，指定有竞争关系的负载运行在相同的成员节点上，也可以避免成员之间的竞争。当然这种方式与应用设计的关系比较密切，需要因地制宜。

10.5 同城双活集群介绍

DB2 集群采用的是共享存储的方式，所以通常 DB2 pureScale 集群都是架建在同一地理位置。但是这种情况下，DB2 pureScale 集群部署在同一站点，如果这个站点整体发生故

障，数据库集群将不可用。

针对这种站点级别高可用的需求，DB2 提供了一种特殊的地理上分散的 DB2 pureScale 集群架构解决方案，也就是 GDPC。所以 GDPC 只是搭建 DB2 pureScale 集群的一种方式，逻辑上还是一套 pureScale 集群，但是物理上分散在不同地理位置。这样即使其中一个站点整体故障，DB2 pureScale 集群也只会损失一半的节点，还有一半的节点存活，仍然可以提供数据库服务。

下面通过图 10-6 来了解 GDPC 同城双活集群的架构：

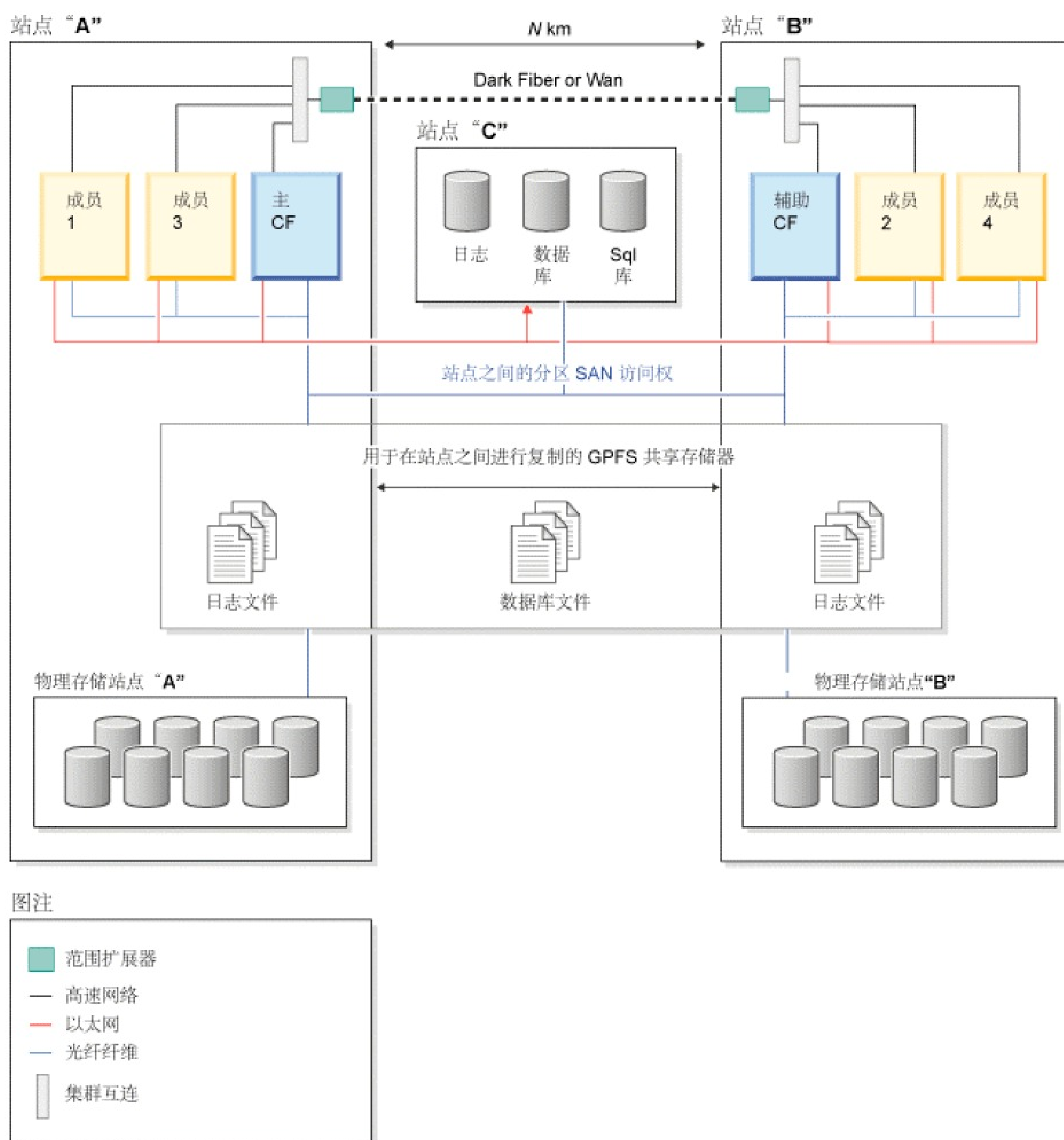


图 10-6　同城双活集群的架构

如上图所示，典型的 GDPC 架构是三站点的方式。站点 A 和站点 B 是对等站点，真正提供数据库服务。站点 C 是仲裁站点，当站点 A 和站点 B 之间网络出现问题的时候，也就是有可能出现所谓的脑裂的情况下，谁抢到了站点 C，谁就是存活的站点。这样的仲裁方法可以避免脑裂的发生。

站点 A 和站点 B 都需要具备独立存活的能力。所有集群里的两个 CF 会分布在这两个站点上，集群里的所有 member 必须是双数，同时必须对半分布在这两个站点上。每个站点都有自己的本地存储，两个站点间存储需要进行实时同步复制。这样即便一个站点出现问题，另外一个站点的存储还是能够一样使用。所以这两个站点几乎可以称为是完全对称的。

站点 C 作为仲裁站点，平时是不工作的，所以没有特殊的性能需求。它只会在出现需要作仲裁的时候起作用。站点 C 也需要设置本地存储，加入到 GPFS 集群里面，作为仲裁盘，不需要放置任何数据，所以对性能也没有要求。

站点 A 和站点 B 之间需要使用 RDMA 协议通信，还需要存储复制。这些功能都由 DB2 pureScale 来实现，但是也需要硬件的支持。所以站点 A、B 之间的网络、存储的 SAN 网络需要相通。这是相对于传统 pureScale 本地集群，实现 GDPC 代价比较高的地方。A、B 站点的距离和网络带宽，会直接影响到数据库的性能。这也是 GDPC 环境最需要关注的地方。

10.6 同城双活集群调优

DB2 GDPC 同城双活集群相对于同机房的 DB2 pureScale 集群，最大的不同是存储需要通过 GPFS 在双中心同步复制，成员节点需要跨站点与 CF 节点通信。相差 70 公里的两个节点，网络 ping 的延时是 0.8 毫秒；而通过 GPFS 复制，写入文件系统的 IO 响应时间是 4 毫秒。这和同机房的 DB2 集群相比是非常大的。所以在 DB2 集群中容易碰到的性能问题在双活环境中更加明显。

下面从存储和通信两个方面介绍如何在 DB2 GDPC 同城双活环境中调优。

10.6.1 减少存储影响

DB2 GDPC 集群访问内部 GPFS 文件系统，读是从本地存储读取，写是双写，需要等到双中心存储都返回成功才算完成。所以对存储的调优主要是在写这方面。DB2 需要写的文件主要是数据文件、日志文件和诊断文件。

对于数据文件，最不希望看到的是同步写盘。所以需要设置较大的本地缓冲池和组缓冲池，同时调整 page_age_trgt_mcr 和 page_age_trgt_gcr 参数。

对于日志文件，数据库的日志是先写入日志缓冲区，然后在提交的时候写入磁盘。所以设置较大的日志缓冲区，能够缓冲更多的日志，提交的时候一次写入更大的 IO 到磁盘。这个缓冲区够用就好，不要出现日志缓冲区满的情况就可以。

对于诊断文件，例如 db2dump 对应的目录，不要放在 GPFS 文件系统里，而是在本地磁盘上建立单独的文件夹来存放。

10.6.2　减少通信影响

DB2 集群内部的通信主要是缓冲池页面的注册、存放和锁资源的申请。减少通信的影响就是从这些方面着手，减少通信开销，尤其是跨站点成员与 CF 的通信。

1. 避免全表扫描

全表扫描会对扫描的所有页面向 CF 节点注册组缓冲池，访问的数据量越大，这种通信开销就越大。所以要创建合适的索引，减少注册组缓冲池的成本。

2. 就近跑批

批处理一般会处理大量数据，不可避免地与 CF 通信。在 DB2 pureScale 集群内部，主 CF 节点和备 CF 节点并非完全一致，只有必要的数据才会一致。所以在观察 CF 资源使用的时候，主 CF 节点的 CPU 真实使用率会高一些。也就是说，成员节点和主 CF 的通信内容更多。批处理一定要放在和主 CF 同机房的成员节点上。

3. 就近 runstats

收集统计信息需要扫描表，这对于成员和 CF 的通信也是比较大的压力。所以请把手动 runstats 工作放在和主 CF 同机房的成员节点上。自动 runstats 也是可控的，在 DB2 集群内部，第一个启动的成员会承担这些管理工作，也就是说，自动 runstats 会运行在这个节点上。所以在集群启动时，先启动和主 CF 同机房的成员节点。

4. 就近访问成员节点

应用服务器通过 DB2 客户端程序或驱动访问集群内部的成员，要避免跨站点访问。所以建议采用客户端偏好的方式，指定同机房应用服务器访问本机房成员节点。

10.6.3　热点表调优案例

热点数据访问的效率问题在 DB2 GDPC 同城双活环境中反映更明显。国内某金融行业公司在部署 GDPC 同城双活集群的时候就遇到了非常麻烦的热点表性能问题。这种表的用

法很常见，就是流水表，存在大量的并发插入。与此同时，该表还存在对插入数据的查询和更新。

```
    CREATE TABLE "CHGMDB "."SERVICE_LOG"  (
                  "ID" INTEGER NOT NULL ,
                  "CLIENT_SERIAL_NUMBER" CHAR(32 OCTETS) NOT NULL ,
                  "CREATE_TIMESTAMP" TIMESTAMP ,
    ...
                  "REQUEST_MESSAGE" CLOB(1048576 OCTETS) INLINE LENGTH 3010
NOT LOGGED NOT COMPACT ,
                  "RESPONSE_MESSAGE" CLOB(1048576 OCTETS) INLINE LENGTH 3010
NOT LOGGED NOT COMPACT
                 ORGANIZE BY ROW;
    CREATE UNIQUE INDEX "CHGMDB  "."SERVICE_LOG_CLIENT_SERIAL_NUMBER" ON
"CHGMDB  "."SERVICE_LOG"
                 ("CLIENT_SERIAL_NUMBER" ASC)
                 INCLUDE NULL KEYS ALLOW REVERSE SCANS;

    CREATE INDEX "CHGMDB  "."SERVICE_LOG_CREATE_TIMESTAMP" ON "CHGMDB
"."SERVICE_LOG"
                 ("CREATE TIMESTAMP" ASC)
                 INCLUDE NULL KEYS ALLOW REVERSE SCANS;

    CREATE INDEX "CHGMDB  "."SERVICE_LOG_ID" ON "CHGMDB  "."SERVICE_LOG"
                 ("ID" ASC)
                 INCLUDE NULL KEYS ALLOW REVERSE SCANS;
```

在没有调优之前，并发插入的性能非常差。通过 db2pd -latches 发现很多的 latch 等待。其中最严重的 latch 是 SQLO_LT_SQLB_BPD__bpdLatch_SX。这个 latch 是热点页的典型特征。这个表里的 ID 列是插入的序列，CLIENT_SERIAL_NUMBER 列是唯一值，SERVICE_LOG_CREATE_TIMESTAMP 也是最怕遇到的热点索引类型。

最初对这张表的调优就是加入了 CURRENT MEMBER 列和将 CURRENT MEMBER 运用到这三个索引里面，也就是把 CURRENT MEMBER 放在索引的第一列。但是这种方式适用的是纯插入场景，对于插入的数据还要查询和更新，效果就打折扣了。调整之后，性能有一定提升，但是离预期的目标相差还很远。同时 CURRENT MEMBER 在索引中的加入也会导致不能保证索引的唯一性。

最后使出了组合拳：

```
    CREATE TABLE "CHGMDB "."SERVICE_LOG"  (
                    "ID" INTEGER NOT NULL ,
                    "CLIENT_SERIAL_NUMBER" CHAR(32 OCTETS) NOT NULL ,
                    "CREATE_TIMESTAMP" TIMESTAMP ,
    ...
                    "REQUEST_MESSAGE" CLOB(1048576 OCTETS) INLINE LENGTH 3010
NOT LOGGED NOT COMPACT ,
                    "RESPONSE MESSAGE" CLOB(1048576 OCTETS) INLINE LENGTH 3010
NOT LOGGED NOT COMPACT ,
                    "CURMEM" SMALLINT IMPLICITLY HIDDEN WITH DEFAULT CURRENT
NODE ,
                    "IDKEY" SMALLINT IMPLICITLY HIDDEN GENERATED ALWAYS AS
(MOD(ID,10) + MOD(CURMEM,4)*10) )
                   PARTITION BY RANGE(IDKEY)
                    (STARTING FROM 0 ENDING AT 39 EVERY 1)
                   ORGANIZE BY ROW;

    CREATE UNIQUE INDEX "CHGMDB  "."SERVICE_LOG_CLIENT_SERIAL_NUMBER" ON
"CHGMDB  "."SERVICE_LOG"
                  ("CLIENT_SERIAL_NUMBER" RANDOM)
                  INCLUDE NULL KEYS ALLOW REVERSE SCANS;

    CREATE INDEX "CHGMDB  "."SERVICE_LOG_CREATE_TIMESTAMP" ON "CHGMDB
"."SERVICE_LOG"
                  ("CREATE_TIMESTAMP" ASC)
                  PARTITIONED
                  INCLUDE NULL KEYS ALLOW REVERSE SCANS;

    CREATE INDEX "CHGMDB  "."SERVICE_LOG_ID" ON "CHGMDB  "."SERVICE_LOG"
                  ("ID" ASC)
                  PARTITIONED
                  INCLUDE NULL KEYS ALLOW REVERSE SCANS;
```

其中 CURMEM 是基于 CURRENT MEMBER 的自生成列，在 4 成员节点的集群里取值是 0 到 3。IDKEY 是基于序列 ID 和自生成列 CURMEM 计算出来的自生成列。方法是用 ID 序列对 10 求余数，结果是 0 到 9，加上 CURMEM 乘 10，最终 IDKEY 的取值是 0-39 循环。成员 0 上生成的 IDKEY 是 0-9 循环，成员 1 上是 10-19 循环，成员 2 上是 20-29 循

环，成员 3 上是 30-39 循环。这种巧妙的设计，使得不同成员生成的数据落在各自独立的分区内部，同时在同一成员内部，相近的数据也循环落在不同的分区内。这样热点数据就被彻底打散了。对于索引，需要保证唯一性的 CLIENT_SERIAL_NUMBER 列采用了 RANDOM 索引，另外两个索引直接使用分区索引。最终调优效果非常明显。

10.7 本章小结

进行 DB2 集群环境性能调优需要对 DB2 集群的内部机制非常了解，在此基础上还要经过一次次的尝试和验证。如同其他调优，这是一个循序渐进、循环处理的过程。通过对监控指标的掌握，了解相应的性能瓶颈，再基于这些优化手段逐步尝试，最终可以达到一个好的效果。

第 11 章 DB2 调优案例、问题总结和技巧

在本章，我们主要给大家举一些案例，希望大家能够通过对这些案例的学习，对性能调优有更深入的认识。这些案例都是来自于国内的某些真实案例，具有很高的参考价值。

本章主要包括如下内容：

- 某移动公司存储设计不当和 SQL 引起的 I/O 瓶颈
- 某银行知识库系统锁等待、锁升级引起性能瓶颈
- 某汽车制造商 ERP 系统通过调整统计信息提高性能
- 某农信社批量代收电费批处理慢调优案例
- 某银行系统系统 SQL 执行慢，通过 track 信息获取调整信息
- 某银行系统字段类型定义错误导致 SQL 执行时间变长
- 某银行客户回单系统 CPU 使用率高
- 某银行手机银行系统 latch 竞争导致 active session 高、性能慢问题
- 利用压力测试程序学习 DB2 性能调优

11.1 调优案例 1：某移动公司存储设计不当和 SQL 引起的 I/O 瓶颈

1. 案例简介

某移动通信公司经营分析系统是一个大型的复杂系统。在这个系统中，从上至下包括

以下几个层次：应用程序、数据库、主机系统(操作系统)、SAN 网络和 ESS 存储系统。在发生系统的性能问题时，性能问题的定位和调优就会很复杂。

2. 问题描述

该移动通信公司经营分析系统从年初开始陆续上线，经营分析系统在上线运行一段时间后出现性能问题，主要表现在对最终用户的交互响应不如预期，尤其在业务繁忙时更是无法得到及时的交互响应。从主机系统上观察，主要表现在系统的 I/O 等待较大。经营分析系统是由业务应用程序、DB2 数据库、AIX 主机、ESS 存储多个部分组成的，因此性能瓶颈的定位和性能的优化都比较复杂。

3. 性能优化步骤

(1) 建立性能调整目标：响应时间能够满足客户要求。

(2) 检查经营分析系统中所有硬件系统，特别是 SAN 网络中的硬件。

(3) 检查 SAN 交换机的数据流量，观察是否有通道流量不对称、数据包丢失或数据传输过程中校验错的问题。

(4) 分析 ESS 上的数据分布，利用相关存储监测软件，观察是否存在 FC 通道、cluster、SSA 卡或 SSA loop 负载不平均的现象。

(5) 检查并优化主机系统上 AIX 运行的参数，使之适合经营分析系统的运行。

(6) 监控数据库运行，检查 DB2 数据库的参数设置是否合理。

(7) 确定最影响性能的应用程序，协助软件开发商优化应用程序。

4. 性能问题的定位、优化措施和建议

1) 主机和存储系统 I/O 优化。

- 优化前后 I/O 资源的使用状况。
- AIX 主机系统观察到的 I/O 资源的使用情况。

以下是 2 月 26 日在 SYSTEMyz2 主机系统上运行 vmstat 得到的结果：

```
kthr     memory      page                        faults    cpu
-------- ----------- -------------------- -------------- -----------
r b p    avm      fre    fi fo pi po fr sr in   sy    cs   us sy id wa
1 0 37   1431905 277052 0  1  0  0  0  0  5362 13902 7791 20  7  2 71
1 0 41   1432744 276179 0  0  0  0  0  0  4559 9019  6271 18  6  2 75
2 0 44   1432881 276038 0  0  0  0  0  0  4414 8652  5838 16  6  1 77
2 0 47   1433726 275189 0  1  0  0  0  0  3998 9811  5112 16  5  1 78
```

```
3 0 45   1432986 275927 0  0  0  0  0  0  4432 13580 6190 21  6  2 71
2 1 44   1433039 275851 2  1  1  0  0  0  4786 10553 6744 17  6  1 76
3 0 39   1433173 275714 0  0  0  0  0  0  5019 11228 7122 22  6  1 71
2 0 34   1433867 275013 0  4  1  0  0  0  3847 18732 4562 14  6  4 76
2 1 32   1431357 277517 0  9  0  0  0  0  4053 10835 5237 18  6  2 74
2 0 39   1431403 277467 0  0  0  0  0  0  4184 18265 5455 19  7  1 74
1 0 43   1431442 277428 0  0  0  0  0  0  4115 9981  5360 14  5  2 79
3 0 40   1434061 274883 0  2  0  0  0  0  4724 16705 6990 22  9  1 69
```

cpu 一列中的 wa(I/O 等待)项均在 70%以上，kthr 列中的 p(pending)项(等待物理 I/O 操作的进程数)达到 40 左右，DB2 数据库端反映对存储设备的操作很慢。以上数据说明存在 I/O 方面的性能问题。

2) I/O 问题的发现与定位

通过如下步骤对 AIX 主机系统和整个系统系统地进行检查：

- 硬件系统检查，包括光纤通道卡，光纤线、SAN 交换机的 FC 通道，以及 ESS 的 FC 通道、硬盘的检查。**最终排除了由于硬件损坏产生的性能影响。**
- SAN 交换机检查，检查分析 SAN 交换机的每个 FC 端口的流量。相应端口流入和流出的数据量相等，没有数据包丢失和 CRC 校验错的情况，**说明 SAN 网络是正常的。**

检查 ESS 的数据分布。根据 ESS 的数据分布图，如图 11-1 所示，可以看出整个经营分析系统有 4 台主机，每台主机有两个卷组。而所有 8 个卷组全部分布在左边 cluster1 的 DA3 和 DA4 两块 SSA 卡上面。产生的结果是所有的 I/O 都集中在 cluster1 的这两块卡的 4 个通道上；而 cluster2 以及其他的 SSA 卡和通道处于空闲状态。**可见，ESS 的数据分布存在问题。通过分析，决定首先应该调整数据在 ESS 上的分布。**

3) 调整方案

首先将数据平均地分布在两个 cluster 上，之后将数据分布在尽可能多的通道上。

由于整个数据容量有约 2120GB，整个数据的迁移需要几十个小时的时间，而在生产系统上是不允许有很长的停机时间进行数据迁移的。

根据多个方案的论证对比，决定采用逻辑卷镜像的方案实施数据迁移。具体的步骤是：先将所有的逻辑卷在目的硬盘上建立镜像、同步数据，再将原硬盘上的镜像部分删除。整个数据迁移工作全部在系统的后台进行，共进行了 60 个小时，才完成所有数据迁移。完成两次数据迁移后的数据分布如图 11-2 所示。

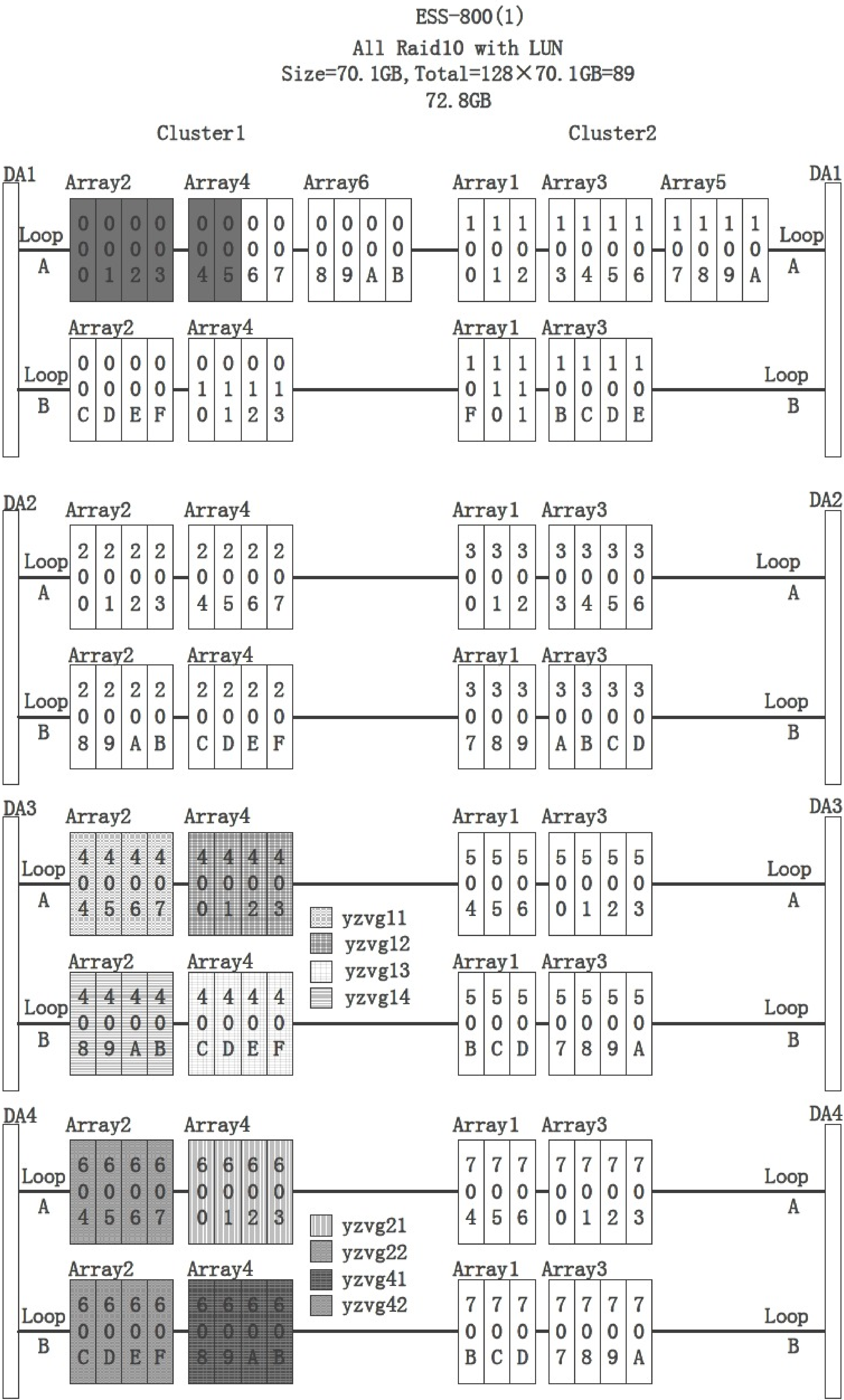


图 11-1　数据分布调整前的 ESS 数据分布图

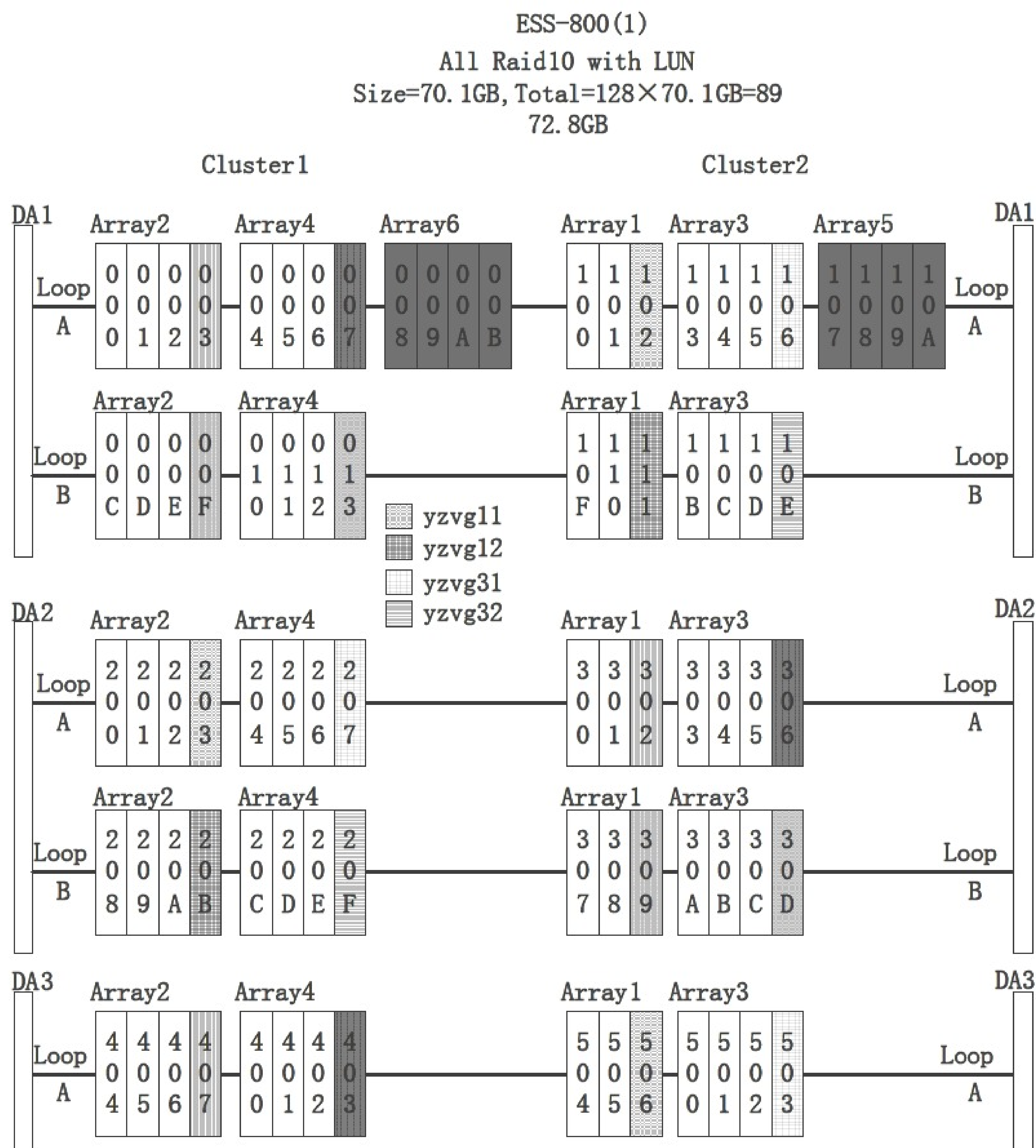


图 11-2　数据分布调整后的 ESS 数据分布图

4) I/O 的调优结果

本次数据迁移的效果非常明显。

以下是 3 月 10 日第二次数据分布调整后，在 SYSTEMyz1 主机系统上运行 vmstat 得到的输出结果：

```
kthr     memory         page                  faults           cpu
-------- -----------  ------------------ ---------------- -----------
r b p   avm     fre      fi fo pi po fr sr  in    sy    cs   us sy id wa
4 0 16 1090229 2944637 0  12 0  0  0  0   3186 27259 2037 48 5  9  38
4 1 15 1090005 2944860 0  11 0  0  0  0   3085 3581  1981 37 4  16 42
5 0 16 1090636 2944227 0  10 0  0  0  0   3300 6748  1953 38 6  17 39
6 0 12 1089367 2945492 0  8  0  0  0  0   3324 8141  2017 38 7  22 34
```

```
3  0  16  1090008  2944851  0  10  0  0  0  0  3608  3297   1921  40  4  7   48
5  0  15  1090007  2944851  0  11  0  0  0  0  3710  3326   1927  39  5  5   51
5  0  12  1089371  2945490  0  12  0  0  0  0  3073  3275   1654  35  5  21  39
3  0  12  1090013  2944848  0  8   0  0  0  0  3329  3675   1932  37  4  26  33
5  0  15  1091313  2943546  0  12  0  0  0  0  3368  5642   2124  39  6  15  40
5  0  17  1090653  2944201  0  8   0  0  0  0  3784  11124  2789  51  8  3   38
```

I/O 等待值由原来的 70%以上下降到了 30%左右，个别时间甚至降至 10%以下。等待物理 I/O 操作的进程数由 40 多个下降到了十几个。但是这仍然存在 I/O 等待。这个时候我们要检查数据库和应用 SQL 问题。

存储调整前资源的使用状况

图 11-3 所示是 3 月 19 日 SYSTEMyz1 主机系统的 I/O 使用率的曲线图。在这段时间内，CPU 的使用率基本上保持在 100%，存在很大的 I/O 等待。

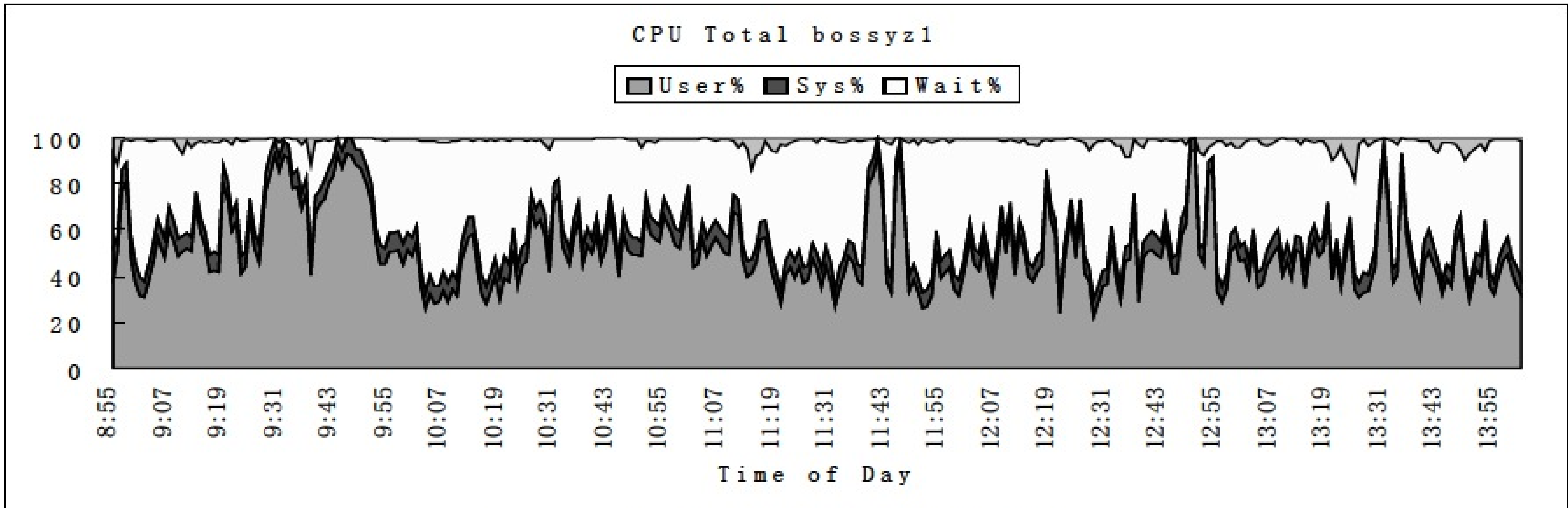


图 11-3　存储调整前资源的使用情况

存储调整后资源的使用情况

根据上面的调整结果，系统 I/O 的状况已经得到了极大改善，如图 11-4 所示，系统的性能瓶颈主要存在于应用对 CPU 资源的消耗。

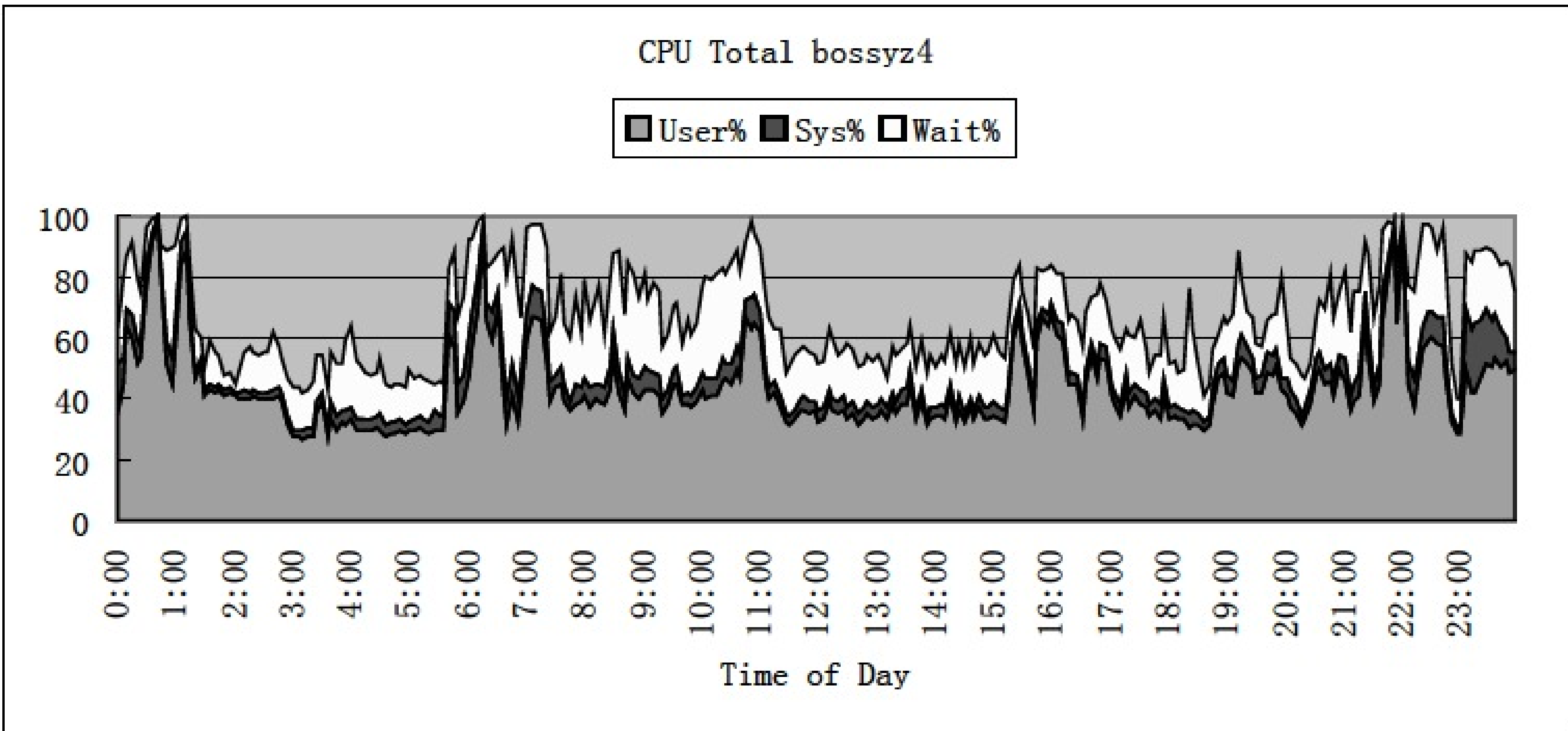


图 11-4　存储资源调整前资源的使用情况

5. 应用和 DB2 数据库的性能分析和建议

利用 DB2 的监控工具，检查系统的配置参数和应用 SQL 语句，发现系统中最消耗资源的前 12 条语句，如图 11-5 所示。

```
$ db2 "select (total_usr_cpu_time+total_sys_cpu_time) as total_cpu_time,substr(stmt_text,1,256) as stmt_text
        from table(SNAP_GET_DYN_SQL_V91('',-2)) as s order by total_cpu_time desc fetch first 12 rows only"
 TOTAL_C
 PU_TIME        STMT_TEXT
 ----- --------------------------------------------------------------------------------------------------
   817 SELECT "CN_LINK_ID" , "CHANGE_TSTAMP" , "CONTRACT_INT" , "OBJECT_TYP" , "OBJECT_ID" , "FUNCTION" ,R
   362 INSERT INTO "BCA_PAYMITEM" VALUES( ? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ? ,?
   288 SELECT * FROM "ZCDM_CARD_PARAM" WHERE "CLIENT" = ? AND "PARAM_NAME" = ?  WITH UR -- OPTLEVEL( 5 )
   249 SELECT "CONTRACT_INT" , "CN_CHANGE_TSTAMP" , "VALID_FROM" , "VALID_TO" , "VALID_TO_REAL" , "CN_BEGD
   239 SELECT "CONTRACT_INT" , "AC_CHANGE_TSTAMP" , "VALID_FROM" , "VALID_TO" , "VALID_TO_REAL" , "ACNUM_C
   232 SELECT "CONTRACT_INT" , "TURNOVER_CLASS" , "BALANCE" , "LCOLDATE" , "LPOSTDATE" , "LAST_REL_TMSP" ?
   223 UPDATE SYSTOOLS.HMON_ATM_INFO SET TO_WAIT = ?, LAST_WAIT = ?, STATS_LOCK = 'N', STATS_STATE = ?, SE
   196 SELECT * FROM "ZCDM_CARD_PARAM" WHERE "CLIENT" = ? AND "PARAM_NAME" = ? AND "PARAM_INDEX" = ?  FET
   166 INSERT INTO "ZCDM_CARD_LOG" VALUES( ? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ? , ?
   166 SELECT "CONTRACT_INT" , "PRENOTE" , "KEY_TIMESTMP" , "PRENOTE_TYPE" , "MEDIUM" , "BUPA_NAME" , "BUF
   164 SELECT * FROM "BP001" WHERE "CLIENT" = ? AND "PARTNER" = ?  WITH UR -- OPTLEVEL( 5 ) -- QUERY_DEGR
   154 SELECT * FROM "ZCDM_TRNS_CHNL" WHERE "CLIENT" = ? AND "CHANNEL" = ?  FETCH FIRST 1 ROWS ONLY OPTIM

   12 record(s) selected.
```

图 11-5　最消耗资源的 12 条 SQL 语句

营账分析系统调优建议

- 检查资源消耗最大语句的执行计划，看看是否有全表扫描。
- 建立合适的索引并对大表考虑使用表分区技术。
- 将排名靠前语句的表数据与索引分别存储。
- 对这些 SQL 语句分别查看执行计划，在统计信息正确的情况下使用 db2advis 工具生成索引建议。

调整后的 I/O 情况如下所示

```
 kthr       memory          page                          faults         cpu
 -------- ----------- ------------------------ ------------ ------------
  r  b  p  avm     fre    fi  fo  pi  po  fr    sr   in    sy   cs us sy id    wa
  6  1  0 1596680 617970 76 179  0  0  103   231 123 21663 5694 47  7 21  24
 10  0  0 1597370 617280  0   0  0  0    0     0 2849 20606 7139 59  4 32  5
  9  0  2 1597496 617153  0 162  0  0    0     0 2899 19424 7701 48  5 35 13
  3  0  2 1596686 617963  0   0  0  0    0     0 2889 19406 8154 56  4 31  9
  9  0  2 1596686 617963  0   0  0  0    0     0 2415 12492 6682 46  4 48  3
 16  0  4 1596688 617961  0   0  0  0    0     0 3621 79477 8622 74  9 11  6
 19  0  0 1597982 616667  0   0  0  0    0     0 2778 20368 6493 74  5 16  6
  2  0  2 1596688 617961  0   0  0  0    0     0 2343 15304 6307 47  3 44  6
  2  0  0 1596692 617956  0   0  0  0    0     0 2792 17646 7537 62  4 26  7
 12  0  6 1598015 616633  0   0  0  0    0     0 3293 25440 6911 69  8 17  6
```

I/O 等待值由原来的 30%以上下降到了 10%左右，个别时间甚至降至 10%以下。等待物理 I/O 操作的进程数由 40 多个下降到了几个。

6. 案例总结

在这个案例中，造成营账分析系统响应时间慢的主要原因是存在大量的 I/O 等待，而造成 I/O 等待的原因有两个：一是因为在最初的存储规划设计时没有正确合理地分布数据；二是数据库中某些复杂的 SQL 语句未创建合理的索引。

经过调整，营账分析系统的速度大大提高，前台客户的响应时间大大减少，可以充分满足客户需求。

11.2 调优案例 2：某银行知识库系统锁等待、锁升级引起性能瓶颈

1. 案例简介

某家银行知识库系统，数据库中出现大量锁等待和锁升级，前台最终用户的响应时间无法满足正常业务需求。

2. 问题描述

主要表现在对最终用户的交互响应不如预期，尤其在业务繁忙时更是无法得到及时的交互响应。从主机系统上观察，主要表现在系统的 CPU 使用比较高，数据库中存在大量的连接，每个连接都需要处理很长时间。

3. 性能优化步骤

(1) 建立性能调整目标：响应时间能够满足客户要求。
(2) 检查系统中所有硬件系统，确认硬件没有问题。
(3) 检查 AIX 内核参数和 DB2 参数设置是否合理。
(4) 监控数据库运行，检查数据库中每个 session 的执行情况。
(5) 监控数据库运行，找到锁升级的表名和锁等待的锁链。
(6) 调整数据库参数或协助优化 SQL 语句，以减少获取锁的数量。

4. 性能问题的定位、优化措施和建议

本案例排查所有硬件环境，都没有发现异常。只能检查 db2diag.log，在数据库诊断日志中发现如下信息：

```
2008-08-31-16.16.17.676115   Instance:db2inst1   Node:000
PID:660232(db2agent (kdbdb) 0)   TID:1   Appid:GB9C4D16.D7CC.011C17D6BBF2
data management  sqldEscalateLocks Probe:3   Database:KDBDB
ADM5502W  The escalation of "19" locks on table "KDB     .O_ORG_INFO" to lock
intent "S" was successful.
2008-08-31-16.16.20.182935   Instance:db2inst1   Node:000
PID:1405022(db2agent (kdbdb) 0)   TID:1   Appid:GB9C4D16.D82C.011C17D7131B
data management  sqldEscalateLocks Probe:3   Database:KDBDB
ADM5502W  The escalation of "21" locks on table "KDB     .TREE" to lock intent
"S" was successful.
2008-08-31-16.16.19.700960   Instance:db2inst1   Node:000
PID:1323556(db2agent (kdbdb) 0)   TID:1   Appid:GB9C4D16.D664.011C17D5A27F
data management  sqldEscalateLocks Probe:3   Database:KDBDB
ADM5502W  The escalation of "18" locks on table "KDB   .O_OPERATOR_INFO" to
lock intent "S" was successful.
2008-08-31-16.16.13.708945   Instance:db2inst1   Node:000
PID:1237448(db2agent (kdbdb) 0)   TID:1   Appid:GB9C4D16.D5AE.011C17D57040
data management  sqldEscalateLocks Probe:3   Database:KDBDB
ADM5502W  The escalation of "3380" locks on table "KDB .TREE" to lock
intent "S" was successful.
2008-08-31-16.14.31.551520   Instance:db2inst1   Node:000
PID:1114452(db2agent (kdbdb) 0)   TID:1   Appid:GB9C4D16.D5B2.011C17D57126
data management  sqldEscalateLocks Probe:3   Database:KDBDB
ADM5502W  The escalation of "3" locks on table "KDB     .UPDATE_TREE" to lock
intent "S" was successful.
```

表 kdb.tree、kdb.update_tree、kdb.O_OPERATOR_INFO 和 kdb. O_ORG_INFO 这 4 张表经常存在 S 锁升级现象，这严重影响数据库的性能。

通过快照监控发现数据库中存在大量锁等待：

```
                Database Lock Snapshot
Database name                            = DEV
Database path                            = /db2/DEV/db2dev/NODE0000/SQL00001/
Input database alias                     = DEV
Locks held                               = 49
Applications currently connected         = 38
Agents currently waiting on locks        = 6
Snapshot timestamp                       = 08-31-2008 15:26:00.951134
Application handle                       = 6
Application ID                           = *LOCAL.db2dev.030815021007
Sequence number                          = 0001
Application name                         = disp+work
Authorization ID                         = SAPR3
Application status                       = UOW Waiting
Application code page                    = 819
Locks held                               = 0
Total wait time (ms)                     = 0
Application handle                       = 97
Application ID                           = *LOCAL.db2dev.030815060819
Sequence number                          = 0001
Application name                         = tp
Authorization ID                         = SAPR3
Application status                       = Lock-wait
Status change time                       = 08-31-2008 15:08:20.302352
Application code page                    = 819
Locks held                               = 6
Total wait time (ms)                     = 1060648
  Subsection waiting for lock            = 0
  ID of agent holding lock               = 100
  Application ID holding lock            = *LOCAL.db2dev.030815061638
  Lock object type                       = Row
  Lock mode                              = Exclusive Lock (X)
  Lock mode requested                    = Exclusive Lock (X)
  Name of tablespace holding lock        = PSAPBTABD
  Schema of table holding lock           = KDB
  Name of table holding lock             = TREE
  Lock wait start timestamp              = 08-31-2008 15:08:20.302356
  输出信息多，略........................
```

1) 问题处理过程

(1) 首先检查数据库配置并进行性能监控

经过查看数据库的配置参数和数据库的性能监控，发现数据库中的锁内存(locklist)参

数为 10240，在监控数据库中发现该值(locklist)的使用高水位已经达到最大值，所以首先调整该参数，把它调整为 20480：

```
db2 update db cfg for kdbdb using locklist using locklist 20480
```

调整后继续监控数据库，发现锁内存的使用高水位仍然达到最大值，所以这个时候怀疑是大量应用没有释放锁才导致即使分配了更多的锁内存，也仍然无法满足要求。这种情况下考虑从应用程序 SQL 入手。

(2) 找出引起锁等待的 SQL 语句

执行 db2 get snapshot for dynamic sql on kdbdb |sort -5 +1 > top.sql grep –E "Total execution time " top.sql > time.sql sort -5 +1 time.sql，发现这样一条 SQL 语句的执行时间竟然花费 22808 秒：

```
Total execution time (sec.ms)      = 22808.2574 秒
```

利用 db2pd 工具找出具体的引起锁等待的 SQL 语句。

要找出发生锁等待时持有哪个锁，我们使用 db2pd 实用程序。为了阅读方便，下面对 db2pd 输出进行了修改，删除除了锁定的行以外的所有行：

```
/home/db2inst1$db2pd -db locktest -locks show detail
Locks:
Address    TranHdl   Lockname                   Type      Mode Sts
 Owner      Dur HldCnt     Att Rlse
0x402C07E0 3         00020002000000000A0000000052 Row       ..X  G
 3          1   0        8   0x40 TbspaceID 2 TableID 2 RecordID 0xA
0x402C02E0 2         00020002000000000A0000000052 Row       .NS  W
 3          1   0        0   0x0  TbspaceID 2 TableID 2 RecordID 0xA
0x402C03A8 2         000200020000000090000000052 Row       ..X  G
 2          1   0        8   0x40 TbspaceID 2 TableID 2 RecordID 0x9
```

可以看到，DB2 持有 TbspaceID 2 表空间中 TableID 2 表上的锁。现在找出这是哪个表：

```
/home/db2inst1 $db2 "select substr(tabschema,1,9) as tabschema,
       substr(tabname,1,12) as tabname, tableid, tbspaceid
       from syscat.tables
       where tbspaceid = 2 and tableid = 2"
TABSCHEMA TABNAME      TABLEID TBSPACEID
--------- ------------ -------- ---------
KDB       TREE             2        2
  1 record(s) selected.
```

上面的 db2pd 输出提供了被锁定的行的记录标识(RID)。值 0xA 实际上表示 0x0000000A，

RID 是由一个三字符页号(这里是 0)和一个单字符 slot 标识(这里是 0xA，也就是 10)组成的 4 字符字段。它告诉我们所关注的这一行是在表的第 0 页的 slot 10 中。每个数据页最多有 255 个“slot”，它包含给定行在页中的偏移量。RID 通常描述为(页号；slot 数)，也就是十进制的(0; 10)、二进制的(0；A)。RID 唯一地标识了表中的一行。

输出表明等待的是 db2pd 输出中的行(0；A)上的锁，因为提供锁请求状态的 Sts 列显示 W，表示等待。其他锁的状态为 G，表示授予(granted)，因此它们被持有。

因此，总结有关的锁：

- 具有 TranHdl 2 的代理：X 锁对于主表行(0；9)为 GRANTED(由于未提交的插入)。
- 具有 TranHdl 3 的代理：X 锁对于主表行(0；A)为 GRANTED(由于未提交的插入)。
- 具有 TranHdl 2 的代理：NS 锁对于主表行(0：A)为 WAITING(由于选择)。

运行 select 语句的代理等待的行是值为 NODE_ID=62481 的行，而不是它刚才更新的行。可以推断由于更新的行上有 X 锁——新更新的行独占性地锁定，直到更新被提交。同时，一个代理不能等待它自己，如果一个代理对于它已经拥有独占(X)锁的行请求共享(NS)锁，那么这个请求会被授予，因为已经拥有了一个具有足够或更高模式的锁。通过 db2pd 定位发现引起锁等待的 SQL 语句如下：

```
    select * from ( select rownumber() over(order by ut.AUDIT DATE DESC) as
rownumber ,ut.MODI ID,ut.NODE ID,ut.NEWNODE TEXT,org.ORG NAME orgname,
borg.ORG NAME borgname, ut.FUNCTION ID, ut.NEWNODE LINK, ut.AUDIT DATE,
t.NODE ID tNodeId
   from
    kdb.update_tree ut left join kdb.O_OPERATOR_INFO op on ut.OP_CODE =
op.OPERATOR_CODE
   left join kdb.O_ORG_INFO org on org.ORG_CODE = ut.ORG_CODE
   left join kdb.O_ORG_INFO borg on borg.ORG_CODE = ut.B_ORG_CODE left join
kdb.TREE ton t.NODE_ID = ut.NODE_ID
   where     op.OPERATOR_ROLE = '9003100004'     and ut.ORG_CODE is not null and
ut.ORG_CODE != ''     and ut.AUDIT_DATE is not null and ((ut.NEWNODE_LINK is not
null and ut.NEWNODE_LINK != '') or ut.KEY_VALUE = '3') and ut.B_ORG_CODE in
('510000000','360000000','650000000','540000000','130000000','120000000','6
20000000','500000000','340000000','150000000','520000000','322000000','6100
00000','210000000','230000000','422000000','212000000','450000000','6300000
00','530000000','640000000','220000000','460000000','140000000','010210000'
,'010210100','010210000') and ut.AUDIT_FLAG = '2' and ut.FUNCTION_ID != '3' and
ut.FUNCTION_ID != '4' and ut.INFO_TYPE concat ut.B_ORG_CODE <> '0010210000' order
```

```
by ut.AUDIT_DATE DESC ) as temp_ where rownumber_ <= ? ;
```

与应用人员交流，发现这条 SQL 语句的执行次数特别多，对该 SQL 语句的执行时间评估：

```
    db2batch -d kdbdb -f 1.sql
```

发现这条 SQL 语句的执行时间大概在 3 秒左右(如果资源紧张，执行时间会更长)，这条 SQL 语句会导致表 kdb.tree、kdb.update_tree、kdb.O_OPERATOR_INFO 和 kdb. O_ORG_INFO 这 4 张表经常发生 S 锁升级现象。在 kdb.tree 发生 S 锁升级后，进而会导致另外一条经常执行的 SQL 语句：

```
    update kdb.TREE set HIT RATIO = HIT RATIO +1 where NODE ID=?(此处为参数标记，
用 62481 实际值代替)
```

出现锁等待现象(locktimeout)，那么在这条语句出现锁等待的阻塞后，进而会影响后续的 SQL 语句：

```
    "select * from ( select rownumber() over(order by ut.AUDIT DATE DESC) as
rownumber ,     ut.MODI ID,ut.NODE ID,ut.NEWNODE TEXT,org.ORG NAME
orgname,borg.ORG NAME
borgname,ut.FUNCTION ID,ut.NEWNODE LINK,ut.AUDIT DATE,t.NODE ID  tNodeId
from    kdb.update tree ut left join kdb.O OPERATOR INFO op on ut.OP CODE =
op.OPERATOR CODE                    left join kdb.O ORG INFO org on org.ORG CODE
= ut.ORG CODE                    left join kdb.O ORG INFO borg on borg.ORG CODE =
ut.B ORG CODE left join kdb.TREE t on t.NODE ID = ut.NODE ID where
op.OPERATOR ROLE = '9003100004'     and ut.ORG CODE is not null and
ut.ORG CODE != ''     and ut.AUDIT DATE is not null     and ((ut.NEWNODE LINK is
not null and ut.NEWNODE LINK != '') or ut.KEY VALUE = '3') and ut.B ORG CODE
in ('510000000','360000000','650000000', '540000000','130000000', '120000000',
'620000000','500000000','340000000','150000000','520000000','322000000','61
0000000','210000000','230000000','422000000','212000000','450000000','63000
0000','530000000','640000000','220000000','460000000','140000000','01021000
0','010210100','010210000') and ut.AUDIT FLAG = '2' and ut.FUNCTION ID != '3'
and ut.FUNCTION ID != '4' and ut.INFO TYPE concat ut.B ORG CODE <> '0010210000'
order by ut.AUDIT_DATE DESC ) as temp_ where rownumber_ <= ? "
```

这样就会出现大量的锁不释放，而连接不释放又导致数据库连接耗尽。这是造成数据库问题的根本原因。

造成问题的主要原因是：由于 SQL 语句执行时间过长，导致这条 SQL 语句引起级联的后续阻塞，这样一来，连接不释放，锁不释放，直到把数据库的连接耗尽和锁内存达到最大值。

2) 问题总结

① 建议如有必要，在这条 SQL 语句的后面加上 with ur 参数，这样该 SQL 语句在读取时就不会在所应用的 4 张表上加 S 锁，从而可以把这个问题解决掉。这需要和业务人员沟通，让他们评估这样做对业务是否有影响。

② 对这条 SQL 语句考虑能否优化逻辑。

③ 利用 db2advis 工具为这条 SQL 语句生成索引建议。

④ 为什么以前没有出现这个问题呢？判断是随着数据库中表的数据量和业务的增多导致的，原来数据量小的时候，这个问题暂时没有表现出来。

⑤ 检查数据库的统计信息是否为最新，否则更新统计信息。

⑥ 检查 DB2_EVALUNCOMMITTED 注册变量。这个设置使 DB2 不必事先在 CS 或 RS 隔离级别锁住一行才判断 SARGable 谓词，这样在我们确定这一行满足谓词之前，它不会锁住。不过，访问未锁定的数据可能会有副作用(比如这个注册变量的设置改变了隔离级别)，不是每个业务都能接受这个副作用的，因此在使用这个功能之前对它加以了解是很重要的。关于该参数的详细描述，参见“第 5 章：锁和并发”。

⑦ 检查 DB2_SKIPINSERTED 注册变量。这个变量控制未提交的插入在 CS 或 RS 隔离级别下是否可以被游标忽略。启用这个变量会使未提交的插入被当成它们完全没被插入一样处理。同样，这种行为也许可以被接受，也许不能被接受，因此了解它的隐含后果很重要，需要和开发人员沟通。关于该参数详细描述，参见“第 5 章：锁和并发”。

⑧ 目前数据库中最影响应用的是如下几条 SQL 语句，分别对这些 SQL 语句分析执行计划，找出消耗资源的操作，并调用 db2advis 进行索引建议：

```
    select * from ( select rownumber() over(order by ut.AUDIT DATE DESC) as
rownumber ,ut.MODI ID,ut.NODE ID,ut.NEWNODE TEXT,org.ORG NAME orgname, borg.
ORG NAME borgname,ut.FUNCTION ID,ut.NEWNODE LINK, ut.AUDIT DATE, t.NODE ID
tNodeId from kdb.update tree ut left join kdb.O OPERATOR INFO op on ut.OP CODE
= op.OPERATOR CODE
    left join kdb.O ORG INFO org on org.ORG CODE = ut.ORG CODE
    left join kdb.O ORG INFO borg on borg.ORG CODE = ut.B ORG CODE
    left join
    kdb.TREE t on t.NODE ID = ut.NODE ID
    where
    op.OPERATOR ROLE = '9003100004'
    and ut.ORG CODE is not null
    and ut.ORG CODE != ''
    and ut.AUDIT DATE is not null
    and ((ut.NEWNODE LINK is not null
    and ut.NEWNODE_LINK != '') or ut.KEY_VALUE
```

```
    = '3') and ut.B ORG CODE in ('510000000', '360000000', '650000000',
'540000000', '130000000','120000000','620000000', '500000000', '340000000',
'150000000', '520000000','322000000','610000000','210000000', '230000000',
'422000000','212000000','450000000','630000000','530000000','640000000','22
0000000','460000000','140000000','010210000','010210100','010210000')
    and ut.AUDIT FLAG = '2'
    and ut.FUNCTION ID != '3'
    and ut.FUNCTION ID != '4'
    and ut.INFO TYPE
    concat ut.B ORG CODE <> '0010210000' order by ut.AUDIT DATE DESC )
    as temp
    where rownumber_ <= ? ;

    select oorginfos0 .PARENT ORG as PARENT2 1 , oorginfos0 .ORG CODE as
ORG1 1 ,oorginfos0 .ORG CODE as ORG1 2 0 , oorginfos0 .PARENT ORG as
PARENT2 2 0 , oorginfos0 .ORG NAME as ORG3 2 0 , oorginfos0 .ORG TYPE as
ORG4 2 0 , oorginfos0 .OP CODE as OP5 2 0 , oorginfos0 .ADMIN CODE  as
ADMIN6_2_0_ from KDB.O_ORG_INFO oorginfos0_ where oorginfos0_.PARENT_ORG=?;

    update kdb.TREE set HIT_RATIO = HIT_RATIO + 1 where NODE_ID=74554 ;
```

通过下面的脚本监控，发现引起锁等待的 SQL 语句主要是以下这 3 条：

```
    select substr(stmt_text,1,127) from
         table(snapshot statement('KDBDB',-1)) a where a.agent id in
            (select agent_id_holding_lk from
                 able(snapshot lockwait('KDBDB',-1)) b );
    -----------------------------------------------------------------
    update TREE set HIT_RATIO = HIT_RATIO + 1 where NODE_ID=54101
    update TREE set HIT RATIO = HIT RATIO + 1 where NODE ID=52372
    select * from ( select rownumber() over(order by ut.AUDIT_DATE DESC)
    as rownumber , ut.MODI ID,ut.NODE ID,ut.NEWNODE TEXT,
    -
    update TREE set HIT RATIO = HIT RATIO + 1 where NODE ID=73152
    select * from ( select rownumber() over(order by ut.AUDIT_DATE DESC) as
rownumber ,      ut.MODI ID,ut.NODE ID,ut.NEWNODE TEXT,
    select * from ( select rownumber() over(order by ut.AUDIT DATE DESC) as
rownumber_,      ut.MODI_ID,ut.NODE_ID,ut.NEWNODE_TEXT,
    select * from ( select rownumber() over(order by ut.AUDIT DATE DESC) as
rownumber_,      ut.MODI_ID,ut.NODE_ID,ut.NEWNODE_TEXT,
```

最后，上述问题通过创建合理的索引(调用 db2advis)和调整部分 SQL 语句的隔离级别得到了完整解决。

5. 调优总结

在这个性能调整的过程中，我们利用了 db2pd、快照监控表函数来监控锁等待。这些工具收集的信息使我们可以理解锁会被持有的原因，这又可帮助我们确定避免不必要的锁定的策略和技术。有了这些信息，任何需要了解锁定的人都可以将这里使用的技术和原则应用到自己的场景中，并得出类似的分析和建议。同时，在找出引起锁等待的 SQL 语句后，我们调用 db2expln、db2exfmt 和 db2advis 工具来分析执行计划并创建合理的索引。

11.3 调优案例 3：某汽车制造商 ERP 系统通过调整统计信息提高性能

1. 案例简介

某汽车制造公司 ERP 系统出现性能瓶颈。

2. 问题描述

主要表现在对最终用户的交互响应不如预期，尤其在业务繁忙时更是无法得到及时的交互响应。用户使用的是 DB2 V8.2.4 数据库，运行在 IBM P670 上，配置 8C16G。

3. 性能优化步骤

(1) 建立性能调整目标：响应时间能够满足客户要求。
(2) 检查系统中所有硬件系统，确认硬件没有问题。
(3) 检查 AIX 内核参数和 DB2 参数设置是否合理。
(4) 监控数据库运行，找到执行时间较长和执行次数较多的 SQL 语句。
(5) 分析调整 SQL 的执行计划，降低执行时间，满足性能目标。

4. 性能问题的定位、优化措施和建议

在这个案例中，我们通过监控发现硬件存储 I/O 设计、操作系统相关配置参数、数据库配置参数等都没有问题，而且 SQL 语句相对比较简单，没有什么可以优化的余地；同时在表上也有相应的索引。乍一看，好像没有什么可以调整的，只能添加硬件资源了。但是通过我们对客户业务逻辑的分析，结合数据库内相关表的情况，我们仔细研究发现该公司数据库中访问最频繁的表 CARS 有如下特征：

- 生产的每一辆汽车，在表中都有相应的一行记录。

- 每辆汽车可以由它的 ID 来标识，因此“ID”是表“CARS”的主键(PK)。
- 此外，表中有一个“STATE”列，表明汽车当前处在制造流程中的哪一步。一辆汽车的制造流程从第 1 步开始，然后是第 2 步、第 3 步、...、第 49 步、第 50 步、第 51 步、...、第 98 步、第 99 步，一直到第 100 步——第 100 步意味着汽车已经完工了。已完工的汽车所对应的行仍然保留在表中，后续流程(例如投诉管理、质量保证等)仍要用到这些行。
- 汽车制造商生产 10 种不同型号(“TYPE”列)的汽车。为了简化问题，在这个示例表中，各种汽车型号命名为 A、B、C、D、...、J。
- 除主键索引(在“ID”列上)外，“STATE”列上有一个索引(“I_STATE”)，“TYPE”列上还有一个索引(“I_TYPE”)。

实际上，“CARS”表包含的列远不止“ID”、“STATE”和“TYPE”。为了保护商业秘密和便于举例，在我们的示例表中没有出现其他列。

上述信息给了我们一些提示，我们可以试图在统计信息上面做些调优工作以提高性能。首先我们再温习一下统计信息(详细内容参见“第 8 章：统计信息更新与碎片整理”)。

1) 分布统计信息的类型——频率(frequency)统计信息和分位数(quantile)统计信息

频率统计信息

假设表 CARS 现在有大约 1 000 000 条记录，不同的型号在表中出现的频率如表 11-1 所示。

表 11-1　表 CARS 中 TYPE 列的频率统计信息

TYPE	COUNT(TYPE)
A	506135
B	301985
C	104105
D	52492
E	19584
F	10123
G	4876
H	4589
I	4403
J	3727

型号为 A 的汽车最受消费者的青睐，因此生产的汽车中大约有 50% 是这种型号。型号 B 和型号 C 仅次于型号 A，分别占所有汽车的 30%和 10%。其他所有型号加在一起仅占 10%。

表 11-1 显示了“TYPE”列的频率统计信息。通过基本统计信息，DB2 优化器只能了解到该表包含 1 000 000 行(表的基数)和 10 种不同的值(型号)，即 A 到 J。如果没有分布统计信息，优化器会认为每种值以相同的频率出现，大约都是出现 100 000 次。而一旦生成关于“TYPE”列的分布统计信息，优化器即可了解每种型号真正的出现频率。因此，优化器清楚各种已有型号出现的不同频率。

优化器使用频率统计信息来计算用于检查相等或不等谓词的过滤因子。例如：

```
SELECT * FROM CARS WHERE TYPE = 'H'
```

分位数统计信息

与频率统计信息不同，分位数统计信息与不同值的出现频率无关，而与表中有多少行小于或大于某个值(或者有多少行介于两个值之间)相关。分位数统计信息提供关于一个列中的值是否聚合(avg、max、sum、min 和 count)的信息。为获得这样的信息，DB2 假定列中的值是按升序排列的，并根据正则行间隔确定相应的值。

我们来看看表 CARS 中的“STATE”列，该列按升序排列。根据正则行间隔，即可确定“STATE”的对应值。表 11-2 显示了 CARS 表中 STATE 列的分位数统计信息。

表 11-2　CARS 表中 STATE 列的分位数统计信息

COUNT(row)	STATE ASC
5479	1
54948	10
109990	21
159885	31
215050	42
265251	52
320167	63

(续表)

COUNT(row)	STATE ASC
370057	73
424872	84
475087	94
504298	100
...	100
1012019	100

由于已完工的汽车仍然没有从 CARS 表中删除，因此状态为 100(＝完工)的汽车数量比所有处于其他状态的汽车总和还多。已完工的汽车占表中所有记录的 50%。

注意：

在实际情况下，已完工的汽车数量甚至还要更多(例如超过 99%)。在稍后的具体例子中可看到这种情况。

表 11-2 显示了“STATE”列的分位数统计信息。有了这种关于有多少行分别小于和大于确定值的信息，优化器即可计算出用于测试小于(小于等于)、大于(大于等于)或介于两值之间的谓词的过滤因子。例如：

```
SELECT * FROM CARS WHERE STATE < 100
SELECT * FROM CARS WHERE STATE BETWEEN 50 AND 70
```

根据已有的分位数统计信息计算出来的过滤因子不是很精确，但即使只收集 20 个值，其误差仍然低于 5%。

2) DB2 优化器对分布统计信息的使用——调优案例

来看一下我们调优的这个案例，在此案例中，DB2 优化器可以使用分布统计信息来更合理地估计过滤因子，以便生成更好的访问计划。

下面这个示例查询从已经定义好的 CARS 表中读取数据。对于表 CARS 中的汽车数据，有以下假设：

- 该表的基数为 1 000 000，也就是说该表包含 1 000 000 行。
- 表中 99.9%的汽车是已经完工(“STATE”列= 100)的，这些汽车的相关信息必须保留，以用于后续处理(投诉管理、质量保证等)。剩下的 1 000 辆汽车目前还处在制造流程中。

- 在该表中，制造商提供的从 A 到 J 的 10 种不同汽车型号(“TYPE”列)几乎以相同的频率出现。
- 为了便于读者做这个实验，我们对客户这个表做了修改，建议大家按照下面的脚本在你的数据库中创建表并导入数据：

```
db2 +c -tvf create table cars.sql---------执行时，把注释去掉
DROP TABLE DB2INST1.CARS;
CREATE TABLE DB2INST1.CARS
(   ID CHAR(13) FOR BIT DATA NOT NULL PRIMARY KEY,
     TYPE CHAR(10) NOT NULL,
     STATE SMALLINT NOT NULL)   ) NOT LOGGED INITIALLY;

INSERT INTO DB2INST1.CARS (ID, TYPE, STATE)
------------使用了表的表达式
WITH  TEMP (COUNTER, ID, TYPE, STATE) AS
(   VALUES (0, GENERATE UNIQUE(), 'A', 100)
     UNION  ALL   SELECT       (COUNTER + 1),       GENERATE UNIQUE(),
       CHR(MOD(INT(RAND() * 1000), 10) + 65),       100    FROM
       TEMP    WHERE       (COUNTER + 1) < 999000 )
SELECT  ID, TYPE, STATE FROM    TEMP;

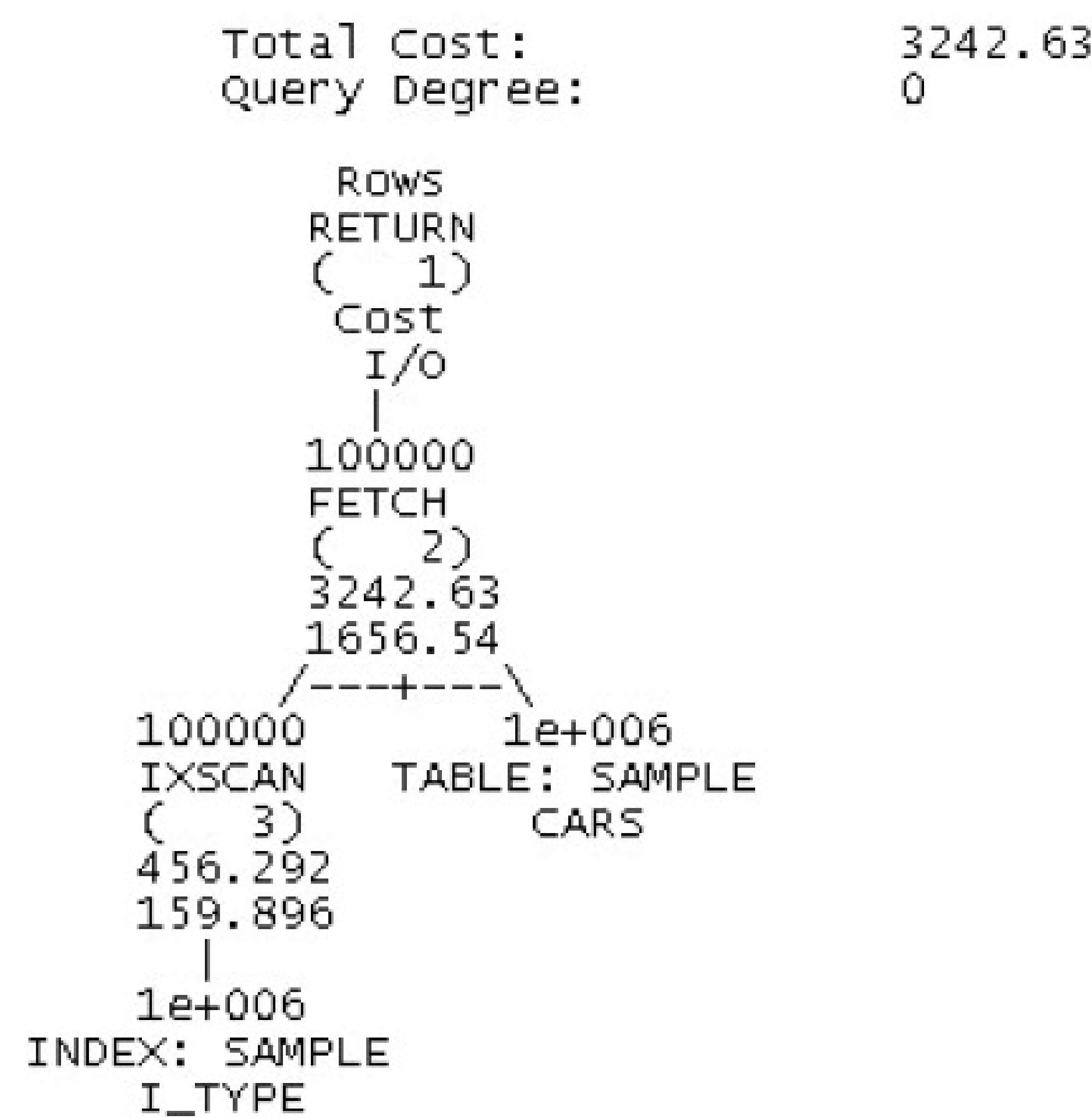
INSERT INTO DB2INST1.CARS (ID, TYPE, STATE)
WITH TEMP (COUNTER, ID, TYPE, STATE) AS
(   VALUES (0, GENERATE UNIQUE(), 'A', 1)
     UNION ALL
     SELECT
       (COUNTER + 1),
       GENERATE UNIQUE(),
       CHR(MOD(INT(RAND() * 1000), 10) + 65),
       MOD(INT(RAND() * 1000), 99) + 1
    FROM
       TEMP
    WHERE
       (COUNTER + 1) < 1000
)  SELECT   ID, TYPE, STATE  FROM    TEMP;//
CREATE INDEX DB2INST1.I TYPE ON DB2INST1.CARS (TYPE ASC) ALLOW REVERSE SCANS;
CREATE INDEX DB2INST1.I STATE ON DB2INST1.CARS (STATE ASC) ALLOW REVERSE
       SCANS;
COMMIT;
```

在他们的业务逻辑中，经常有类似如下的查询，用于选择型号为 A 且正处在制造流程中、尚未完工的所有汽车：

```
SELECT * FROM CARS WHERE STATE < 100 AND TYPE = 'A'
```

首先来分析一下，在没有分布统计信息、而只有 CARS 表的基本统计信息及其索引的

情况下，优化器选择的访问计划是怎样的，如图 11-6 所示。



```
Total Cost:             3242.63
Query Degree:           0

            Rows
           RETURN
           (   1)
            Cost
             I/O
             |
           100000
           FETCH
           (   2)
           3242.63
           1656.54
          /---+---\
     100000        1e+006
     IXSCAN    TABLE: SAMPLE
     (   3)          CARS
     456.292
     159.896
       |
     1e+006
 INDEX: SAMPLE
     I_TYPE
```

图 11-6　没有分布统计信息时查询的访问计划

由于优化器不知道 STATE 值不是均匀分布的，因此它决定使用索引 I_TYPE。使用这个索引会带来较高的成本，因为在访问 CARS 表之前，需要从索引中读取大约 100 000 个 RID(记录 ID)。此外，对于查询返回行数的假设也是错误的。因为优化器认为所有制造步骤(从 1～100)都有相同的频率，所以它无法预知谓词“STATE < 100”将过滤掉大量有价值的行。但是您知道，事实是在所有 1 000 000 辆汽车中，只有 1 000 辆汽车正处于生成流程中。

在没有分布统计信息的情况下，执行该查询时，动态 SQL 的一个快照返回以下监视器值(假定所需的监视器开关已激活)：

```
Number of executions                    = 1
Number of compilations                  = 1
Worst preparation time (ms)             = 9
Best preparation time (ms)              = 9
Internal rows deleted                   = 0
Internal rows inserted                  = 0
Rows read                          = 99336
Internal rows updated                   = 0
Rows written                            = 0
Statement sorts                         = 0
Statement sort overflows                = 0
Total sort time                         = 0
Buffer pool data logical reads     = 8701
Buffer pool data physical reads         = 8131
Buffer pool temporary data logical reads  = 0
```

```
Buffer pool temporary data physical reads  = 0
Buffer pool index logical reads       = 165
Buffer pool index physical reads          = 155
Buffer pool temporary index logical reads  = 0
Buffer pool temporary index physical reads = 0
Total execution time (sec.ms)         = 0.530903
Total user cpu time (sec.ms)              = 0.280403
Total system cpu time (sec.ms)            = 0.230332
Statement text                          = SELECT ID, TYPE, STATE FROM
            SAMPLE.CARS           WHERE STATE < 100 AND TYPE = 'A'
```

在此，我们不会进一步分析这些值，但是请记住它们，以便与有分布统计信息时使用相同查询得到的监视器值相比较。

接下来，为 CARS 表生成分布统计信息，并再次执行查询。此时，优化器选择了图 11-7 所示的访问计划。

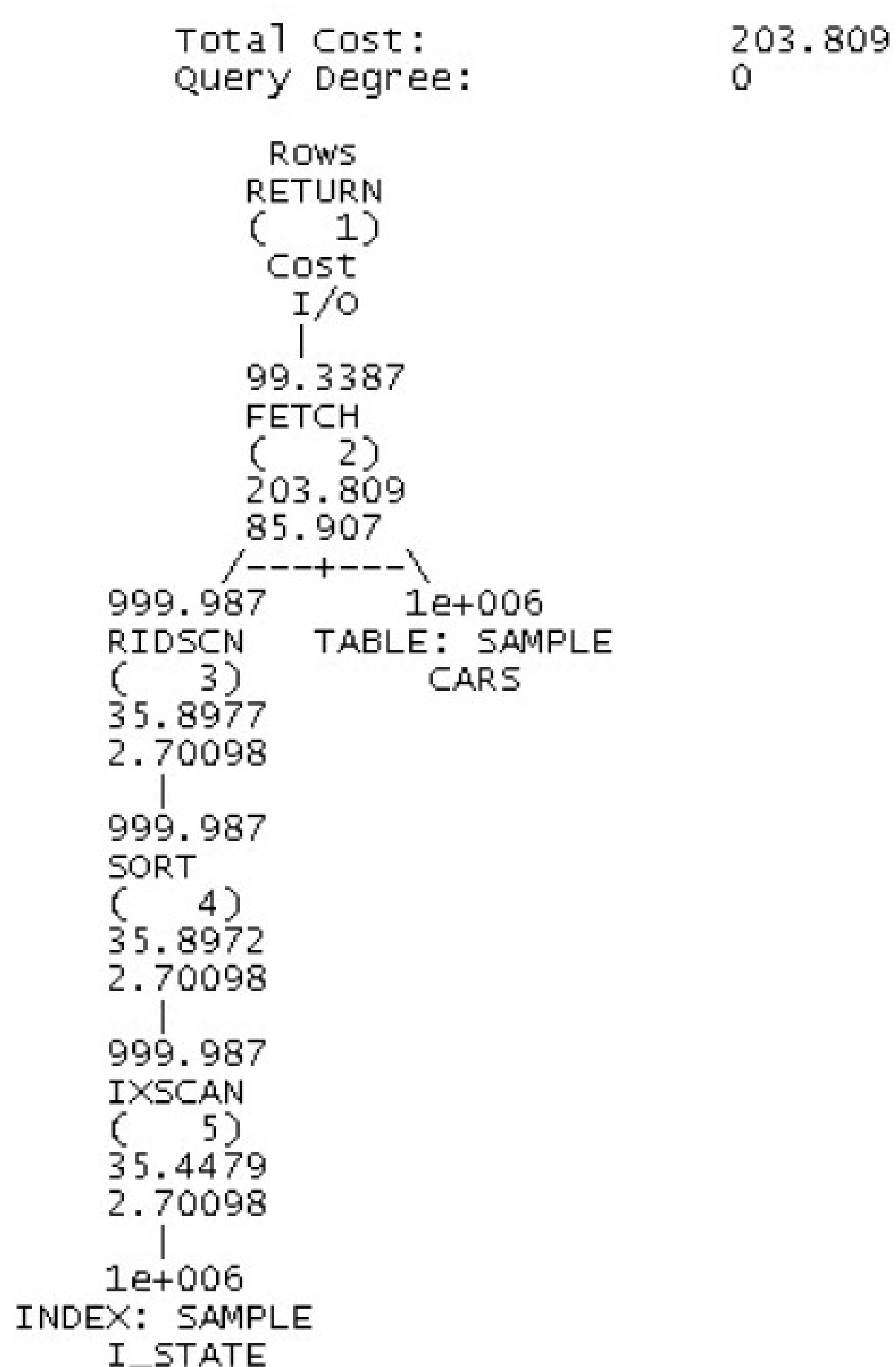


图 11-7　有分布统计信息时查询的访问计划

这个访问计划的成本明显低于没有分布统计信息时的成本：前者为 203.809，而后者为 3242.63。这是因为优化器现在知道，谓词“STATE<100”有较高的过滤因子，因而只会返回大约 1 000 辆正处于生产流程中尚未完工的汽车。因此在这种情况下，CARS 表不是

使用索引 I_TYPE 来访问的，而是使用索引 I_STATE 来访问的。此外，现在可以正确地估计结果集中的总行数。现有 1 000 辆汽车尚未完工，不同的型号出现频率相同。故结果集中包含大约 100 行。

有分布统计信息时的访问计划要优于没有分布统计信息时的访问计划。但是，这是否会影响查询的执行时间？以下是有分布统计信息时查询的监视器数据快照：

```
Number of executions                         = 1
Number of compilations                       = 1
Worst preparation time (ms)                  = 9
Best preparation time (ms)                   = 9
Internal rows deleted                        = 0
Internal rows inserted                       = 0
Rows read                        = 1000
Internal rows updated                        = 0
Rows written                                 = 0
Statement sorts                              = 1
Statement sort overflows                     = 0
Total sort time                              = 5
Buffer pool data logical reads        = 11
Buffer pool data physical reads              = 10
Buffer pool temporary data logical reads    = 0
Buffer pool temporary data physical reads   = 0
Buffer pool index logical reads       = 12
Buffer pool index physical reads             = 9
Buffer pool temporary index logical reads   = 0
Buffer pool temporary index physical reads = 0
Total execution time (sec.ms)         = 0.014597
Total user cpu time (sec.ms)                 = 0.000000
Total system cpu time (sec.ms)               = 0.010014
Statement text                               = SELECT ID, TYPE, STATE FROM
          SAMPLE.CARS                         WHERE STATE < 100 AND TYPE = 'A'
```

表 11-3 比较了有分布统计信息和没有分布统计信息这两种不同情况下的快照监视器值。

表 11-3　比较快照监视器值

快　照　值	无分布统计信息	有分布统计信息
Rows read	99 336	1 000
Buffer pool data logical reads	8 701	11
Buffer pool index logical reads	165	12
Total execution time (sec.ms)	0.530903	0.014597

可以看到，在有分布统计信息的情况下，DB2 执行查询时需要计算的行数更少。这对于 CPU 成本和 I/O 成本都有积极的影响。最重要的是总执行成本，因为总执行成本关系到应用程序的响应时间。在具有分布统计信息的情况下，这个时间是 0.014597 秒，而在没有分布统计信息的情况下，这个时间是 0.530903 秒，相差 36 倍之多。

在我们的示例中，两种情况下的执行时间分别为 0.014597 秒和 0.530903 秒，这个差距还不够明显，因为这两个值只是次秒级的。然而，这样的差距不应被忽略，因为如果这样的查询次数非常多、非常频繁，那么这种提高对性能影响也是非常明显的。同时，如果要执行更复杂的查询，或者要连续执行多个查询，那么执行时间的差距就不是次秒级的，而是以秒甚至分钟来计算的。

分布统计信息的生成

如前所述，在使用 RUNSTATS 命令生成统计信息时，并不是总会收集分布统计信息。这是有意义的，因为仅在存在很多重复值或者数据分布不均匀的情况下，分布统计信息才重要。而在其他情况下，分布统计信息并不能带来多大的好处。

下面的 RUNSTATS 命令只收集 CARS 表(在模式 SAMPLE 中)和相应索引的基本统计信息：

```
RUNSTATS ON TABLE SAMPLE.CARS AND INDEXES ALL
```

此外，如果需要收集 CARS 表中所有列的分布统计信息(频率统计信息和分位数统计信息)，那么可以执行以下命令：

```
RUNSTATS ON TABLE SAMPLE.CARS WITH DISTRIBUTION AND INDEXES ALL
```

生成分布统计信息意味着给 DB2 带来额外、可观的开销，从而影响 RUNSTATS 命令的执行时间。所以，应该只为那些需要分布统计信息的列生成分布统计信息：

```
RUNSTATS ON TABLE SAMPLE.CARS WITH DISTRIBUTION ON COLUMNS (TYPE, STATE)
AND INDEXES ALL
```

应为满足以下条件的列收集分布统计信息：

- 该列有很多重复的值(频率统计信息)，或者该列的值分布不均匀，即它们在某些局部是聚合的(分位数统计信息)。
- 检查等于或不等于谓词中使用到该列(频率统计信息)，或者检查小于(小于等于)、大于(大于等于)或介于两个值之间的谓词中使用到该列(分位数统计信息)。

对于频率统计信息，重要的是定义好收集多少个值的重复数。如果为一个列中的所有值收集频率统计信息，那么成本就太高了。如果在执行 RUNSTATS 时没有显式定义数量，那么 DB2 将使用由数据库参数 num_freqvalues 提供的默认数量。由于 NUM_

FREQVALUES 的默认值为 10，DB2 将为列中出现频率最高的 10 个值收集重复次数，这里假定 RUNSTATS 是在没有显式定义数量，并且数据库参数 num_freqvalues 没有被修改的情况下执行的。

与频率统计信息类似，也必须为分位数统计信息定义数量以保证精确性。分位数统计信息定义应该使用多少“度量值”(measurement)。列中的值被认为是按升序排列的，并且有正则的行间隔，相应的值是确定的。使用的度量值越多，优化器对于检查范围(<、>、<=、>=、BETWEEN)谓词的过滤因子的估计就越准确。如果在执行 RUNSTATS 时没有明确指定值，那么 DB2 将使用由数据库参数 num_quantiles 提供的默认数量。NUM_QUANTILES 的默认值是 20，也就是说使用 20 个度量值。这已经是较好的值，因为可以保证优化器在使用分位数统计信息的情况下对确定过滤因子的估计误差最大只有 5%。

如果数据库配置 db cfg 不能提供 num_freqvalues 和 num_quantiles 的值，那么可以在执行 RUNSTATS 时显式定义：

```
    RUNSTATS ON TABLE SAMPLE.CARS WITH DISTRIBUTION ON COLUMNS (TYPE
num freqvalues 10 num quantiles 20, state num freqvalues 15 num quantiles 30)
AND INDEXES ALL
```

如何检查是否存在分布统计信息

为检查某个表的分布统计信息是否已收集，可以查看分类视图 SYSCAT.COLDIST 的内容：

```
    SELECT * FROM SYSCAT.COLDIST WHERE TABSCHEMA = 'DB2INST1' AND TABNAME = 'CARS'
```

视图 SYSCAT.COLDIST 结构如表 11-4 所示。

表 11-4　SYSCAT.COLDIST 视图的结构

列　　名	数 据 类 型	是否可以为空	描　　述
TABSCHEMA	VARCHAR(128)	不可以	本条目对应的表的限定符
TABNAME	VARCHAR(128)	不可以	本条目对应的表的名称
COLNAME	VARCHAR(128)	不可以	本条目对应的列的名称
TYPE	CHAR(1)	不可以	F = Frequency(最大频率) Q = 分位数值
SEQNO	SMALLINT	不可以	如果 TYPE = F，那么该列中的 N 表示第 N 频繁的值 如果 TYPE = Q，那么该列中的 N 表示第 N 个分位数值
COLVALUE	VARCHAR(254)	可以	数据值，形式为字符字面值或是 NULL 值

(续表)

列　　名	数 据 类 型	是否可以为空	描　　述
VALCOUNT	BIGINT	不可以	如果 TYPE = F，那么 VALCOUNT 是 COLVALUE 出现在该列中的次数 如果 TYPE = Q，那么 VALCOUNT 是其值小于或等于 COLVALUE 的行的数量
DISTCOUNT	BIGINT	可以	如果 TYPE = Q，那么该列记录小于或等于 COLVALUE 的不同值的数量(如果没有，那么为 NULL)

仅在收集了表中至少一个列的分布统计信息时，SYSCAT.COLDIST 才会包含关于该表的条目。如果在没有 WITH DISTRIBUTION 的情况下再次执行 RUNSTATS，那么 SYSCAT.COLDIST 中与该表对应的条目将被删除。

关于分布统计信息的详细描述，请参见“第 8 章：统计信息更新与碎片整理”。

5. 案例总结

在这个案例中，硬件存储 I/O 设计、操作系统相关配置参数、数据库配置参数等都没有问题，而且 SQL 语句相对比较简单，没有什么可以优化的余地；同时在表上也有相应的索引。乍一看，好像没有什么可以调整的，只能添加硬件资源了。但是通过我们对客户业务逻辑的分析，结合数据库内相关表的情况，对业务相关的表做了分布统计信息，从而使性能得到了极大提高，所以大家需要深入理解 DB2 中的每个概念，这样才能结合应用运用地得心应手。

11.4　调优案例 4：某农信社批量代收电费批处理慢调优案例

1. 案例简介

某农信社批量代收电费应用，该应用每天晚上从供电公司 FTP 批量文件过来，有 35 万条记录。处理过程是这样：每读出批量文件的每一行，写入表 batdtl，根据表 batdtl 的数据生成后台批量文件，后台处理完成后返回结果，然后读后台结果文件，更新表 batdtl 中相应记录的状态。

2. 问题描述

问题就出在更新表 batdtl 相应记录的状态这里，35 万条记录做了十几个小时都没做完。

3. 性能优化步骤

(1) 建立性能调整目标：响应时间能够满足批处理要求。
(2) 监控数据库运行，检查 DB2 数据库的参数设置是否合理。
(3) 分析执行计划，调整 SQL 和索引。

4. 性能问题的定位、优化措施和建议

在这个案例中，引起性能慢的 SQL 语句已经通过业务逻辑发现，使用嵌入 SQL 的编程语句：

```
EXEC SQL
UPDATE batdtl SET rec stat=:rec stat, flag=:FLAG, hosterr=:hosterr,
       hostmsg=:hostm
WHERE bat_no=:bat_no AND pay_seq=:seqno;
```

首先执行下列 SQL 语句，查看该表和索引的统计信息是否为最新：

```
db2 "select stats time from syscat.tables where tabname='BATDTL'"
db2 "select stats_time from syscat.indexes where tabname=' 'BATDTL'"
```

经过检查，发现表和索引的统计信息是最新的。

执行：

```
db2expln -d pbranch -q "UPDATE batdtl SET rec stat=:rec stat, flag=:FLAG,
hosterr=:hosterr, hostmsg=:hostm WHERE bat no=:bat no AND pay seq=:seqno" -o
sqlplan.txt
```

注意：

读者在自己的系统上运行时，注意把宿主变量替代成真实的值。

查看该 SQL 语句的执行计划。经过查看，发现该表的 bat_no 和 pay_seq 列有两个单列索引。SQL 语句虽然也走索引扫描了，但是走这种索引扫描 Index ANDing 肯定不是最优的。执行计划如下所示：

```
==================== STATEMENT ==========================================
 Isolation Level         = Cursor Stability
 Blocking                = Block Unambiguous Cursors
 Query Optimization Class = 5
 SQL Path ="SYSIBM","SYSFUN","PBRANCH""SYSPROC","SYSIBMADM",
Statement:
   update pbranch.batdtl set rec stat='00', flag='1',   hosterr='AAAAAAA'
  where bat_no='0210022314'and   pay_seq='1110023846000000'
```

```
Section Code Page = 1386
Estimated Cost = 0.225233
Estimated Cardinality = 0.023810
Index ANDing
|  Optimizer Estimate of Set Size: 1
|  Index ANDing Bitmap Build Using Row IDs
|  |  Access Table Name = PBRANCH.BATDTL  ID = 2,6
|  |  |  Index Scan:  Name = PBRANCH.BATDTL_IND2  ID = 2
|  |  |  |  Regular Index (Not Clustered)
|  |  |  |  Index Columns:
|  |  |  |  |  1: BAT NO (Ascending)
|  |  |  #Columns = 0
|  |  |  #Key Columns = 1
|  |  |  |  Start Key: Inclusive Value
|  |  |  |  |  |  1: '0210022314'
|  |  |  |  Stop Key: Inclusive Value
|  |  |  |  |  |  1: '0210022314'
|  |  |  Index-Only Access
|  |  |  Index Prefetch: None
|  |  |  Isolation Level: Uncommitted Read
|  |  |  Lock Intents
|  |  |  |  Table: Intent None
|  |  |  |  Row   : None
|  Index ANDing Bitmap Probe Using Row IDs
|  |  Access Table Name = PBRANCH.BATDTL  ID = 2,6
|  |  |  Index Scan:  Name = PBRANCH.BATDTL IND1  ID = 3
|  |  |  |  Regular Index (Not Clustered)
|  |  |  |  Index Columns:
|  |  |  |  |  1: PAY_SEQ (Ascending)
|  |  |  #Columns = 0
|  |  |  #Key Columns = 1
|  |  |  |  Start Key: Inclusive Value
|  |  |  |  |  |  1: '1110023846000000'
|  |  |  |  Stop Key: Inclusive Value
|  |  |  |  |  |  1: '1110023846000000'
|  |  |  Index-Only Access
|  |  |  Index Prefetch: None
|  |  |  Isolation Level: Uncommitted Read
|  |  |  Lock Intents
|  |  |  |  Table: Intent None
|  |  |  |  Row   : None
Insert Into Sorted Temp Table  ID = t1
|  #Columns = 1
```

```
    |  #Sort Key Columns = 1
    |  |  Key 1: (Ascending)
    |  Sortheap Allocation Parameters:
    |  |  #Rows     = 1.000000
    |  |  Row Width = 20
    |  Piped
    |  Duplicate Elimination
    List Prefetch Preparation
    |  Access Table Name = PBRANCH.BATDTL  ID = 2,6
    |  |  #Columns = 12
    |  |  Avoid Locking Committed Data
    |  |  Evaluate Block/Data Predicates Before Locking Committed Row
    |  |  Fetch Using Prefetched List
    |  |  |  Prefetch: 1 Pages
    |  |  Lock Intents
    |  |  |  Table: Intent Share
    |  |  |  Row  : Next Key Share
    |  |  Sargable Predicate(s)
    |  |  |  #Predicates = 2
    |  |  |  Return Data to Application
    |  |  |  |  #Columns = 14
    Return Data Completion
    End of section
```

利用：

```
    db2advis -d pbranch -q "UPDATE batdtl SET rec stat=:rec stat, flag=:FLAG,
hosterr=:hosterr, hostmsg=:hostm WHERE bat_no=:bat_no AND pay_seq=:seqno"
```

生成索引建议，结果如下：

```
    execution started at timestamp 2008-10-20-12.05.31.124231
    Recommending indexes...
    total disk space needed for initial set [   1.013] MB
    total disk space constrained to         [ 324.569] MB
    Trying variations of the solution set.
    Optimization finished.
      1  indexes in current solution
     [23000.0000] timerons  (without recommendations)
     [ 75.0000] timerons  (with current solution)
     [99.67%] improvement
    -- LIST OF RECOMMENDED INDEXES
    -- ===========================
    -- index[1],    1.013MB
       CREATE INDEX "DB2INST1"."IDX810201705560000" ON "PBRANCH "."BATDTL"
```

```
("BAT NO" ASC ,"PAY SEQ" ASC) ALLOW REVERSE SCANS ;
       COMMIT WORK ;
       RUNSTATS ON TABLE "PBRANCH "."BATDTL" FOR INDEX
"DB2INST1"."IDX810201705560000" ;
       COMMIT WORK ;
    -- RECOMMENDED EXISTING INDEXES
    -- ============================
    -- UNUSED EXISTING INDEXES
    -- ============================
    -- DROP INDEX "PBRANCH "."BATDTL IND1";
    -- DROP INDEX "PBRANCH "."BATDTL IND2";
    -- ===========================
    7 solutions were evaluated by the advisor
    DB2 Workload Performance Advisor tool is finished.
```

创建索引后，这个批量代收电费应用的批处理时间从 10 个多小时减至 7 分钟。这是一次极佳的性能提高！

5. 案例总结

在这个案例中，造成批量代收电费批处理慢的主要原因是为该 SQL 语句创建了不合理的索引。如果您对创建索引没有把握的话，有一个比较偷懒的办法——利用 db2advis 工具，它会为 SQL 生成最合理的索引。注意应用前一定确保表和索引的统计信息为最新的。

11.5　调优案例 5：某银行系统字段类型定义错误导致 SQL 执行时间变长

1. 案例简介

某银行系统在测试环境数据库中执行某条语句耗时非常短，在生产环境上线新业务之后发生语句执行变慢。

2. 问题描述

在数据量不变的前提下，在测试环境中只需要几秒钟就执行完的 SQL 语句，在硬件性能更强的生产环境却需要数十分钟。

3. 性能优化步骤

(1) 建立性能调整目标：达到测试环境下的执行速度。

(2) 检查系统中的所有硬件系统，确认硬件没有问题。
(3) 检查 AIX 内核参数和 DB2 参数设置是否合理。
(4) trace 该 SQL 语句，发现异常的函数调用。
(5) 调整表结构，降低执行时间，满足性能目标。

4. 性能问题定位、优化措施和建议

通过应用日志，可以发现出问题的 SQL 如下：

```
select count(1)
  FROM FRP.T40 IAMDEALS  TI
    LEFT JOIN FRP.T40 CORPO TW
    ON   TI.CPTY ID = TW.HOST CUST ID
         with ur;
```

出问题 SQL 的执行计划如下：

```
Access Plan:
-----------
        Total Cost:              9.37059e+06
        Query Degree:            1

              Rows
             RETURN
             (   1)
              Cost
               I/O
               |
                1
             GRPBY
             (   2)
           9.37059e+06
              4927
               |
           1.37658e+10
             >NLJOIN
             (   3)
           8.55782e+06
              4927
         /-----+------\
      20772          662709
     TBSCAN          IXSCAN
     (   4)          (   5)
```

```
        224.049         5183.35
          213          4713.22
           |              |
         20772          662709
     TABLE: FRP      INDEX: FRP
      T40 IAMDEALS    P T40 CORPO UN
          Q2              Q1
```

通过查看上述执行计划，看不出什么太大的问题，我们对有问题的 SQL 做 trace，trace 出来的结果如下：

```
    3853          0.002489669   | sqlriChar2GraphicWithLen entry [eduid 100151
eduname db2agent]
    3854          0.002490220   | | sqlriCastC2G entry [eduid 100151 eduname
db2agent]
    3855          0.002490759   | | | sqlriLobBufferManager entry [eduid 100151
eduname db2agent]
    3856          0.002490992   | | | sqlriLobBufferManager exit
    3857          0.002491517   | | | sqlnls char2graph entry [eduid 100151
eduname db2agent]
    3858          0.002491806   | | | | sqlocpcv entry [eduid 100151 eduname
db2agent]
    3859          0.002492289   | | | | | sqlnlscpcv entry
    3860          0.002492521   | | | | | sqlnlscpcv data [probe 5]
    3861          0.002493263   | | | | | sqlnlscpcv data [probe 6]
    3862          0.002493787   | | | | | sqlnlscpcv data [probe 7]
    3863          0.002494281   | | | | | sqlnlscpcv data [probe 11]
    3864          0.002494791   | | | | | sqlnlscpcv data [probe 13]
    3865          0.002495285   | | | | | | sqlnlsCodePageConvert entry
    3866          0.002495490   | | | | | | | sqlnlsUnicodeConv entry
    3867          0.002496376   | | | | | | | sqlnlsUnicodeConv exit
    3868          0.002496582   | | | | | | sqlnlsCodePageConvert data [probe 15]
    3869          0.002497046   | | | | | | sqlnlsCodePageConvert exit
    3870          0.002497494   | | | | | sqlnlscpcv data [probe 990]
    3871          0.002497990   | | | | | sqlnlscpcv data [probe 999]
    3872          0.002498484   | | | | | sqlnlscpcv exit
    3873          0.002498687   | | | | sqlocpcv exit
    3874          0.002499138   | | | sqlnls char2graph exit
    3875          0.002499371   | | sqlriCastC2G exit
    3876          0.002499835   | sqlriChar2GraphicWithLen exit
```

可以看到有很多调用 sqlriChar2GraphicWithLen 的方法，所以怀疑是在做 VARGRAPHIC 的转型操作，进一步检查表结果后，发现 T40_IAMDEALS 表的 CPTY_ID 字段被错误地

定义成 VARGRAPHIC，导致在连接的时候需要做大量的转型操作，降低了执行速度，同时影响执行计划的生成。

解决方法是让开发人员将 VARGRPAHIC 改为 VARCHAR，这样就避免了每次对比都进行转型操作，重建表并且重新导入数据后该 SQL 的执行时间明显缩短。

5. 案例总结

DB2 提供了很强大的 stack 和 trace 功能，可以让我们详细了解到 DB2 的函数调用层次和执行时间。虽然我们不是 DB2 的开发人员，但是有些很难一眼看出来的 SQL 性能问题，仍然可以通过 stack 或 trace 出来的函数名和时间，发现一些蛛丝马迹。

11.6 调优案例 6：某银行客户回单系统 CPU 使用率高

1. 案例简介

客户回单系统负责全行所有交易的回单生成和回单打印功能，可以提供前台界面和 Web Service 接口两种访问方式，客户可以通过手机银行、网络银行、柜台等多渠道进行回单的查询和打印，每天的打印量在百万笔，属于白天比较繁忙的系统。

2. 问题描述

该系统在某次应用版本变更后，在第二天白天业务高峰时段，数据库的 CPU 使用率升高到 90%以上，正常变更前，数据库的 CPU 使用率只有 20%左右，前台业务响应时间明显加长，如图 11-8 所示。

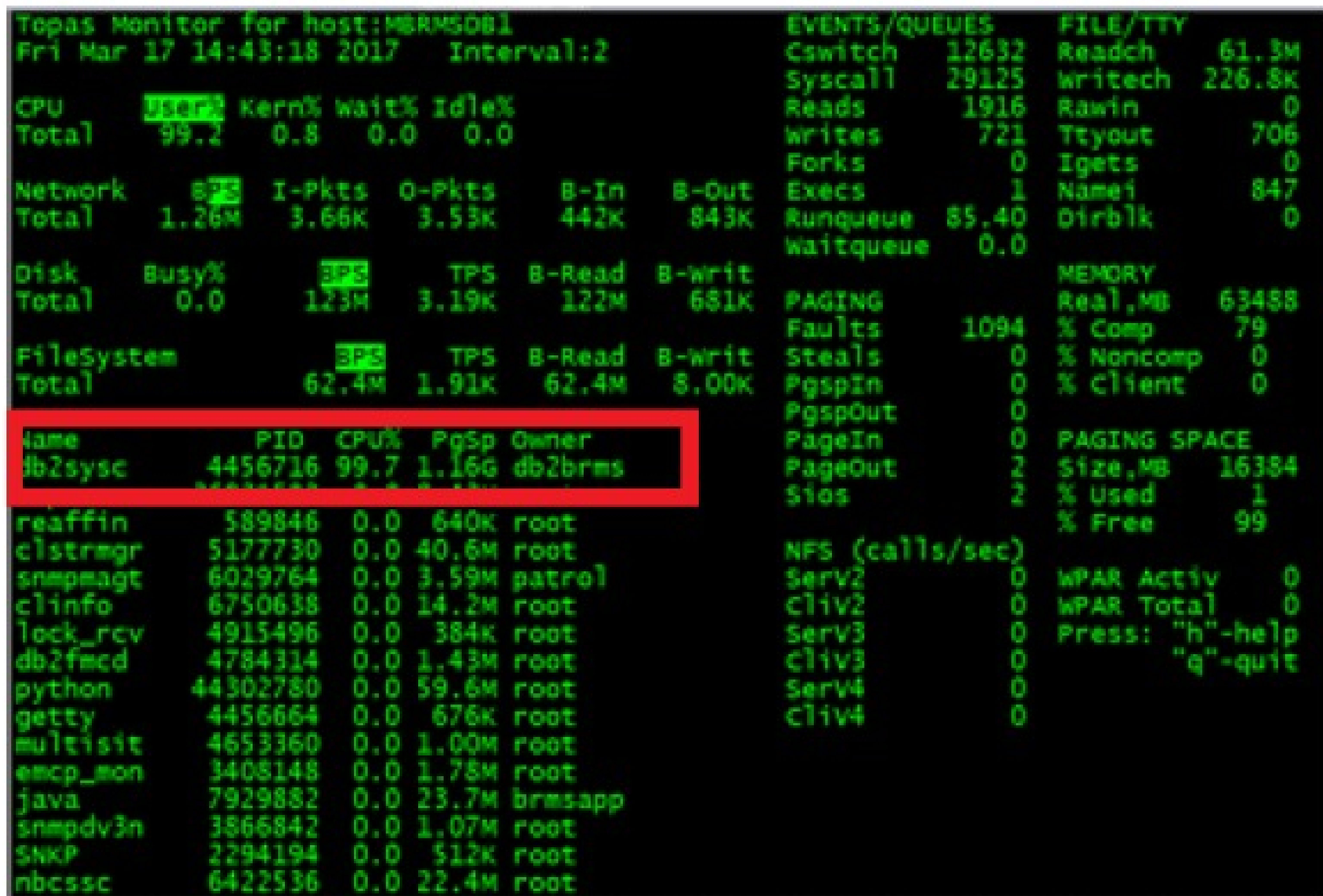


图 11-8 topas 的执行结果

3. 性能优化步骤

(1) 建立性能调整目标：CPU 使用率下降到 30%以下，响应时间能够满足客户要求。

(2) 确定造成 CPU 使用率高的进程。

(3) 检查 DB2 的活动 session 数和每秒扫描最多的表。

(4) 优化或干预 SQL 语句，协助软件开发商优化应用程序。

4. 性能问题的定位、优化措施和建议

出现这种问题后，我们的第一反应就是查看每个进程的 CPU 占用率，通过 topas，可以看到 db2sysc 这个数据库实例进程占用了 99.7%的 CPU。进而我们切换到实例用户，查看 db2top，看 db2 的具体执行情况。按 d 看数据库的整体情况，ActSess 代表当前正在执行的 SQL 数量，77 个正在执行的 SQL，已经明显高于平均数了，如图 11-9 所示。再按 D，看具体每个连接的执行情况，也可见都处于 UOW Executing 状态，如图 11-10 所示。再按 T，看每个表的读写情况，可见 BILL_PRINT_CORP 表每秒的行读数非常高，如图 11-11 所示。

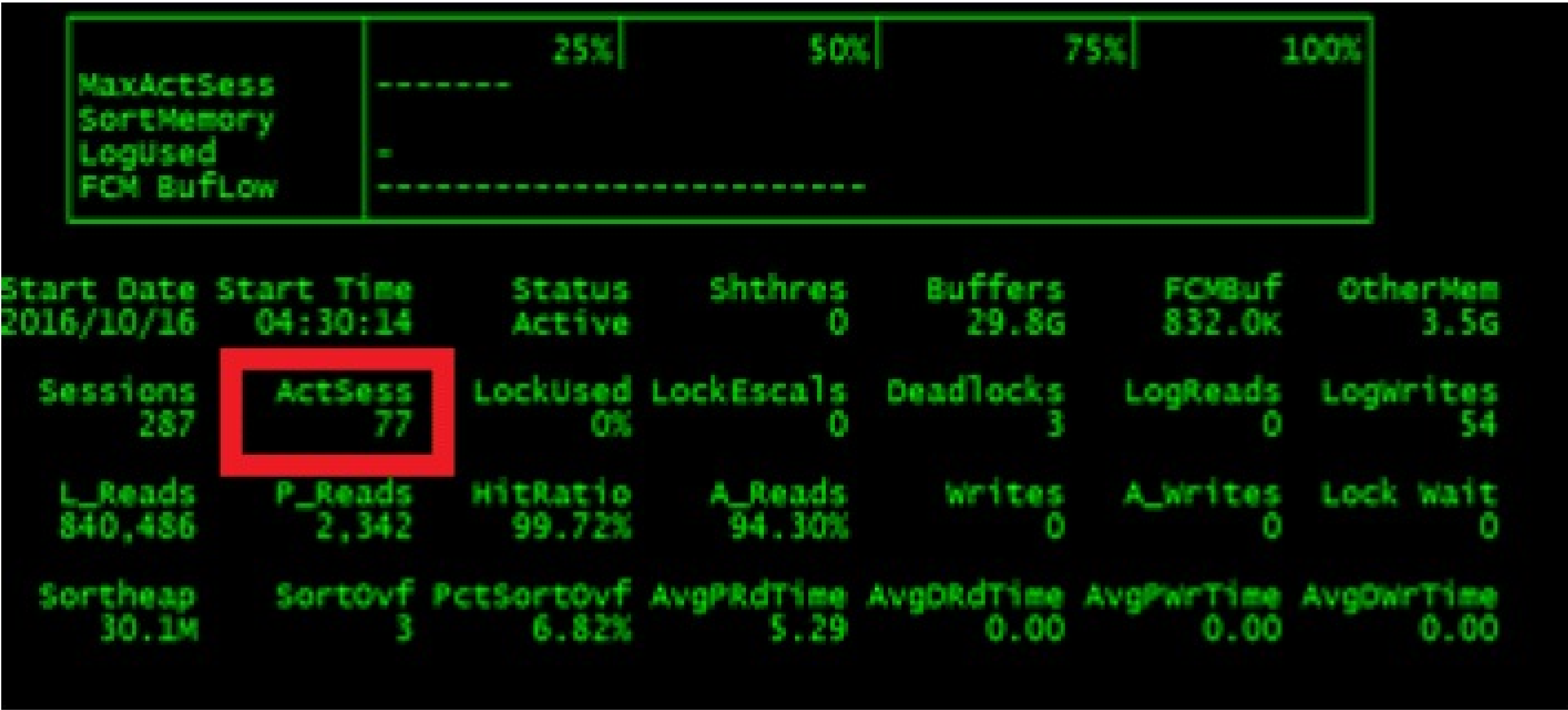


图 11-9 db2top 的执行结果

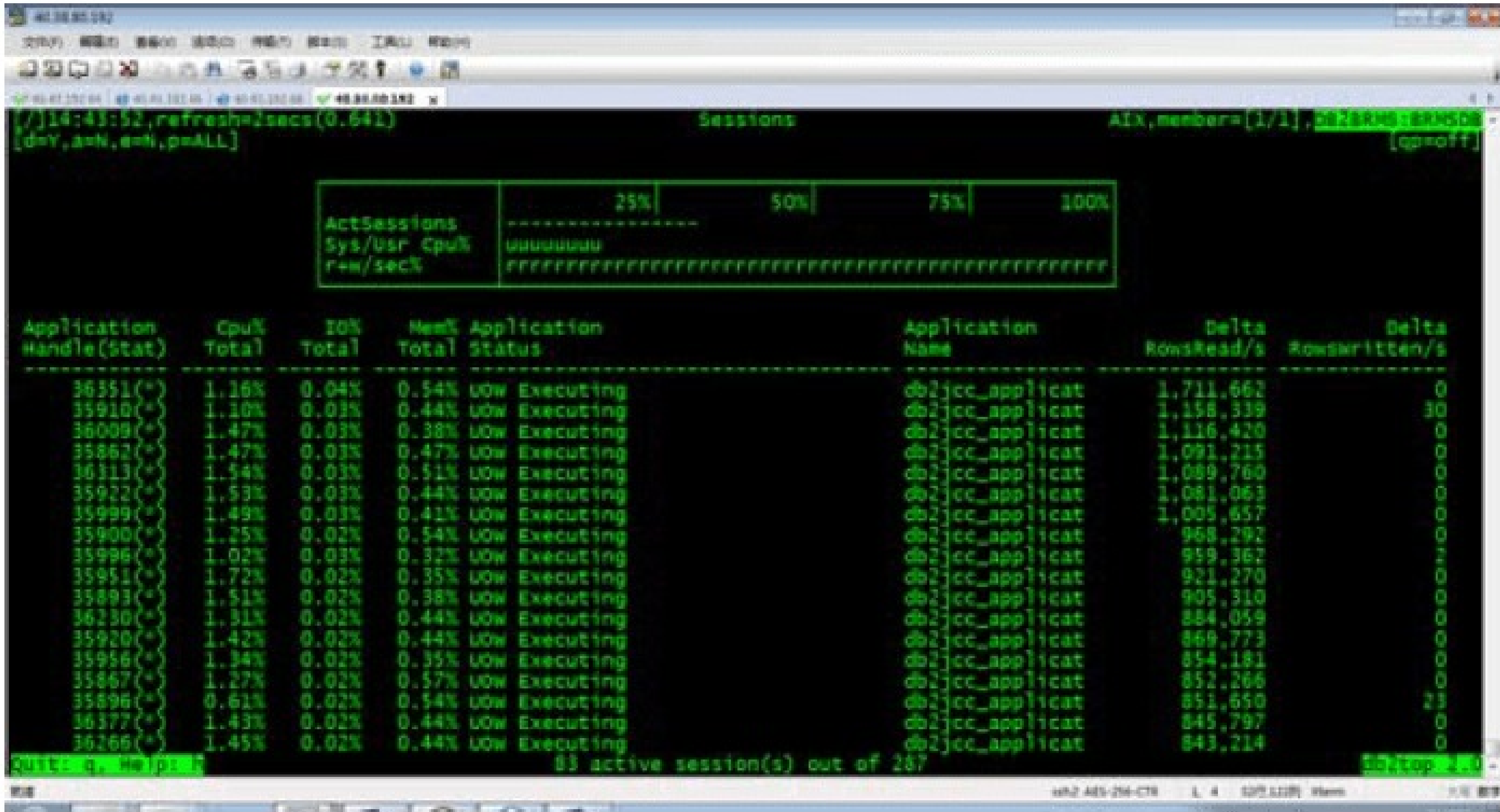


图 11-10 db2top 按 D 的执行结果

图 11-11　db2top 按 T 的执行结果

看到这种情况，就要分析是否有执行了表扫描的 SQL 语句，那么我们立刻抓取所有 agent 的 snapshot：

```
db2 get snapshot for applications on brmsdb
```

查看行读较多的 SQL，发现图 11-12 所示 SQL 的行读较多。

图 11-12　行读较多的 SQL

```
select
  *
from
  (select
     rownumber() over(order by
                     billflowco0_.SA_ACCT_NO,
                     billflowco0_.SA_TX_DT,
                     billflowco0_.INNER_NO) as rownumber_,
```

```
        billflowco0 .INNER NO as col 0 0 ,
        。。。 。。。
        billprintc1 .IS PATCH as col 46 0 ,

      case
      when billprintc1 .PRINT CNTS is null
      then 0
      else billprintc1 .PRINT CNTS end as col 47 0
      from
        BILL FLOW CORP billflowco0
        left outer join BILL PRINT CORP billprintc1
        on billflowco0 .INNER NO=billprintc1 .INNER NO
      where
        1=1 and
        (billflowco0 .SA ACCT NO='56567676767' or
         billflowco0 .SA ACCT NO='56786543434' or
         billflowco0 .SA ACCT NO='12346747' or
         billflowco0 .SA ACCT NO='7854456331' or
         billflowco0 .SA ACCT NO='4564345678' or
         billflowco0 .SA ACCT NO='789765678644' or
         billflowco0 .SA ACCT NO='2345645323456' or
         billflowco0 .SA ACCT NO='56756754433355' or
         billflowco0 .SA ACCT NO='12343234543' or
         billflowco0 .SA ACCT NO='687978473234' or
         billflowco0 .SA ACCT NO='385764324567' or
         billflowco0 .SA ACCT NO='3785739683' or
         billflowco0 .SA ACCT NO='48596843245' or
         billflowco0 .SA ACCT NO='284896059322') and
        billflowco0 .ACCT TYP='2' and
        billflowco0 .PRINT CNTS=0 and
        billflowco0 .IS PATCH=' '
      order by
        billflowco0 .SA ACCT NO,
        billflowco0 .SA TX DT,
        billflowco0 .INNER NO
     ) as temp
   where
     rownumber_ <= ?
```

BILL_FLOW_CORP 表是回单的流水主表，有 4 亿条数据，表上有 INNER_NO 的索引。BILL_PRINT_CORP 表是回单的附属表，记录每条回单的打印次数，有 1 亿条数据，也以 INNER_NO 作为索引。理论上，如果先通过账号在回单流水主表里过滤出少部分流水数据，

然后再通过索引关联打印次数，也不会出现 CPU 高的问题，那么我们就用 db2exfmt 命令来查看这个 SQL 的执行计划，发现 DB2 对打印次数表错误地选择了表扫描的方式，就是这种错误的执行计划造成了该 SQL 执行速度低下、CPU 耗用高，参见图 11-13。

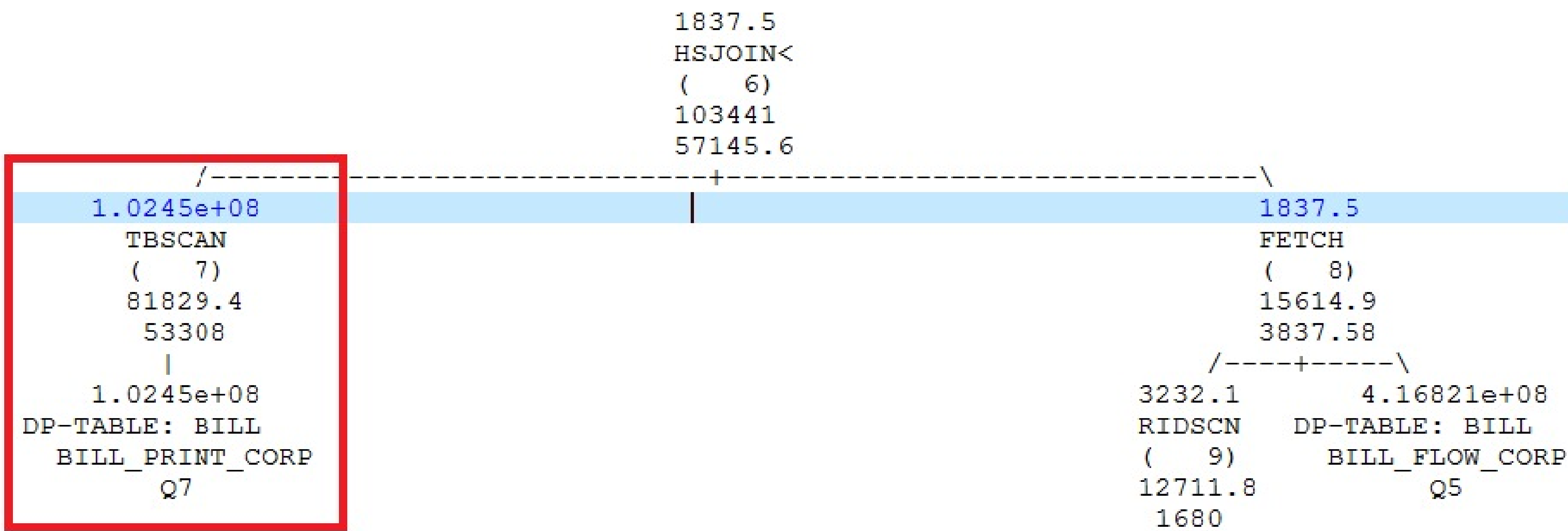


图 11-13　优化前的执行计划

找到问题后，我们需要分析一下问题产生的原因。首先看统计分析，都是最新的，没有什么问题，因此出问题的原因很有可能是 DB2 计算了一个错误的成本，因此我们采用 DB2 的语句级的 OPTGUIDELINES 来干预执行计划。在该 SQL 语句的后面增加如下优化方案，要求 DB2 对该 SQL 只使用最小查询优化，强制该 SQL 使用走索引的 NLJOIN。

```
/*<OPTGUIDELINES> <QRYOPT VALUE="0"/> </OPTGUIDELINES>*/
```

优化后的执行计划如图 11-14 所示，打印次数表通过索引和回单流水主表进行 NLJOIN。开发更新完代码，重新部署生产后，目前日间 CPU 只要 20%不到了，并且客户的响应时间大大降低。

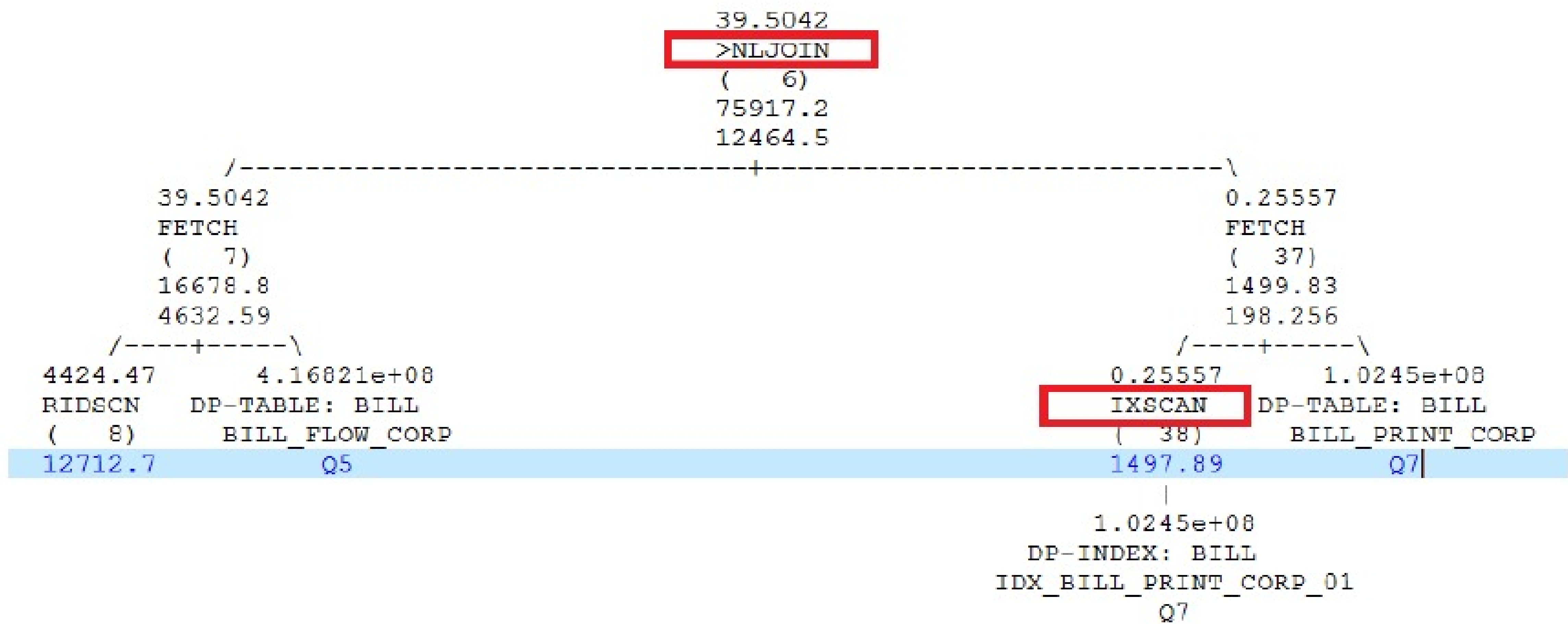


图 11-14　优化后的执行计划

5. 案例总结

通过该案例，希望读者可以了解 SQL 调优的三板斧：

(1) 看活动会话的个数和表扫描最多的表。

(2) 找到表扫描最多的表对应的 SQL 语句。

(3) 通过 OPTGUIDELINES 来干预执行计划。

有时候，DB2 优化器不一定有你想象的那么智能，太复杂的 SQL 很可能会干扰 DB2 优化器的判断，去选择不应该使用的执行计划。这个时候，就需要通过一些办法来纠正这种错误，比如：

(1) 运行更细粒度的统计分析。

(2) 通过 OPTGUIDELINES 来强制修改执行计划。

所以碰到这种问题不要慌张，按照上面的步骤做，一定能够找到解决的办法。

11.7 调优案例 7：某银行手机银行系统 latch 竞争导致 active session 高、性能慢问题

1. 案例简介

手机银行系统是最重要的渠道系统之一，不论白天还是晚上都比较繁忙，查询业务量和办理业务量一直比较大。某一天突然出现手机银行登录缓慢问题，严重影响客户的正常使用。

2. 问题描述

该系统在问题出现之前的一段时间内没有应用程序的变更，也没有数据库上的变更，只是在交易量上有平稳的增长。问题发生时，数据库服务器的 CPU 使用率升高到 90%以上，活动会话(active session)达到 2000 以上，其中一部分处于 commit active 状态，同时通过 db2pd 看到有大量的 latch 等待。

3. 性能优化步骤

与之前的案例不同，这是一个紧急的性能问题，需要在很短的时间内解决，因此相比之前的案例多了若干应急处理的过程。

为了尽快恢复业务服务水平，针对数据库服务器 CPU 高和 active session 高的现象，采取了重启 DB2 实例的措施，但重启之后发现 active session 几分钟之内再次超过 2000，问

题没有解决。

重启数据库实例不生效，只能按部就班进行分析和优化：

(1) 建立性能调整目标：降低 active session，降低 CPU 使用率。

(2) 分析 active session 当前正在执行的任务和堵塞的原因。

(3) 分析 latch 的含义，寻找解决该 latch 的方法。

4. 性能问题的定位、优化措施和建议

该问题发生时，数据库监控系统第一时间发出了数据库服务器 CPU 使用率高和 active session 高的告警。我们马上通过 topas 确认了数据库实例进程 db2sysc 占用了 97%的 CPU。切换到实例用户，用 db2top，按 d 查看数据库的整体情况，看到 ActSess 有 2000 多，但其他指标看起来都正常；进一步按 l(小写字母)，看到有大量的 session 处于 commit active 状态。

基于以上信息，怀疑存储设备有故障而导致数据库 commit 操作无法完成。但经过确认，此前存储链路有过 5 秒钟的中断，但已经恢复正常。

进一步使用 db2pd -latch 重复多次查看 latch 情况，可以看到主要是由一个名为 xhshlatch 的 latch 导致的等待。部分结果如下：

```
   Address              Holder     Waiter     Filename                  LOC
LatchType          HoldCount
   0x0A00020001DA22A8       511780                                    69051
/view/db2_v97fp10_aix64_s141015/vbs/engn/sqp/inc/sqlpLockInternal.h     513
SQLO_LT_SQLP_LHSH__xhshlatch 1
   0x0A00020001DA22A8       511780                                   261703
/view/db2_v97fp10_aix64_s141015/vbs/engn/sqp/inc/sqlpLockInternal.h     513
SQLO_LT_SQLP_LHSH__xhshlatch 1
   0x0A00020001DA22A8       511780                                   401212
/view/db2_v97fp10_aix64_s141015/vbs/engn/sqp/inc/sqlpLockInternal.h     513
SQLO_LT_SQLP_LHSH__xhshlatch 1
   0x0A00020001DA22A8       511780                                    52797
/view/db2_v97fp10_aix64_s141015/vbs/engn/sqp/inc/sqlpLockInternal.h     513
SQLO_LT_SQLP_LHSH__xhshlatch 1
   0x0A00020001DA22A8       511780                                    72509
/view/db2_v97fp10_aix64_s141015/vbs/engn/sqp/inc/sqlpLockInternal.h     513
SQLO_LT_SQLP_LHSH__xhshlatch 1
   0x0A00020001DA22A8       511780                                   125054
/view/db2_v97fp10_aix64_s141015/vbs/engn/sqp/inc/sqlpLockInternal.h     513
SQLO_LT_SQLP_LHSH__xhshlatch 1
```

```
   0x0A00020001DA22A8       511780                            621642
/view/db2_v97fp10_aix64_s141015/vbs/engn/sqp/inc/sqlpLockInternal.h      513
SQLO_LT_SQLP_LHSH__xhshlatch 1
   0x0A00020001DA22A8       511780                            102770
/view/db2_v97fp10_aix64_s141015/vbs/engn/sqp/inc/sqlpLockInternal.h      513
SQLO_LT_SQLP_LHSH__xhshlatch 1
```

由于 latch 是 DB2 内部的一种机制，能看到有 latch 等待，但还是不知道 DB2 是在处理什么操作的过程中遇到了 latch。这时候我们收集 stack，使用的命令是“db2pd -stack all”。通过分析 stack，发现主要有两个 stack：

```
   Stack 1:
   <StackTrace>
   -------Frame------ ------Function + Offset------
   0x090000002104B258 thread_waitlock@glue881 + 0x8C
   0x090000002104AFE0 sqloXlatchConflict + 0x1E8
   0x0900000020F59820 sqloXlatchConflict@glue1AB@clone0 + 0x7C
   0x0900000020F59900 sqlplrm__FP8sqeAgent + 0x24
   0x0900000020F8E518 sqlpxcm1__FP8sqeAgentP15SQLXA_CALL_INFOi + 0x46C
   0x0900000020F8D49C sqlrrcom_dps__FP8sqlrr_cbiT2P15SQLXA_CALL_INFO + 0x174
   0x0900000020F8AC4C sqlrrcom__FP8sqlrr_cbiT2 + 0x398
   0x0900000020F8B500 sqlrr_commit__FP14db2UCinterface + 0xDC
   ......

   Stack 2:
   <StackTrace>
   -------Frame------ ------Function + Offset------
   0x090000002104B258 thread_waitlock@glue881 + 0x8C
   0x090000002104AFE0 sqloXlatchConflict + 0x1E8
   0x090000002104AD3C sqloXlatchConflict@glue1AB + 0x78
   0x0900000020F3D3C4 sqlplrq__FP9sqeBsuEduP14SQLP_LOCK_INFO + 0x24
   0x0900000020F85FBC sqlrr_get_lock__FP8sqlrr_cbP14SQLP_LOCK_INFOUiPUi +
0x100
   0x0900000020F85DD0
sqlra_pkg_lock__FP8sqlrr_cbPUcsT2T3T2iUiT7P14SQLP_LOCK_INFO +
   0xB80x0900000020F852C4
sqlra_find_pkg__FP8sqlrr_cbPUcsT2T3T2T3iT8P14SQLP_LOCK_INFOPP20sqlra_cached
_package + 0x190
   0x0900000020F84ADC sqlra_revalidate_pkg__FP8sqlrr_cb + 0xA0
   0x0900000020F84E24 .sqlra_load_pkg.fdpr.clone.2194__FP8sqlrr_cbPUcsT2T3T2b
```

```
+ 0x190
   0x0900000020F7DB3C sqlrr_sql_request_pre__
FP14db2UCinterfaceUiiP16db2UCprepareInfoP15db2UCCursorInfo + 0x1678
   ......
```

从 stack 的函数名可以大致知道，第一个 stack 是在作 commit 操作时释放 lock，对应 active session 中正在 commit active 的那部分 session，第二个 stack 是在对 package 进行加锁操作。

结合以上信息进行分析，手机银行登录时需要在数据库上执行一系列语句，每次操作在数据库内需要获取同一个 package lock，并在 commit 时释放该 package lock。因为是同一个 lock，需要操作同一个 lock hash table(DB2 内存中的一种内部数据结构)。latch xhshlatch 用于顺序化多线程访问 lock hash table，高并发的登录操作导致对 xhshlatch 的争用，从而导致登录响应慢。

知道了问题的原因之后，开始寻找如何避免 latch 的方法。经过与 IBM 技术支持工程师联系，建议将 DB2_APM_PERFORMANCE 设为 ON。设置该变量后重启数据库实例，问题暂时得以解决。

DB2 知识中心提到，“将 DB2_APM_PERFORMANCE 设为 ON，会启用无包锁定方式。此方式允许全局查询高速缓存运行，而不必使用包锁定，这些锁定是内部系统锁定，可以保护高速缓存的包条目不会被除去。NO PACKAGE LOCK 方式可能会使性能有较小的提高，但不允许执行某些数据库操作。这些被禁止的操作可能包括：使包无效的操作、使包不起作用的操作、PRECOMPILE、BIND 和 REBIND。”

尽管说是“有较小的提高”，但是在处理该手机银行的问题时，由于去掉了 package lock 机制，从而避免了 xhshlatch，所以在该问题的处理上该参数发挥了巨大的作用。同时也提到将 DB2_APM_PERFORMANCE 设置为 ON 有副作用，即一些与 package 相关的操作无法执行。后来与 IBM 技术支持工程师进一步沟通，在 DB2 V10.5 中该参数的值有更多的选择，可以将其设置为“16”，基本没有副作用。

事后，在 IBM 网站上找到了一篇 technote——“Slow performance with latch contention on SQLO_LT_SQLP_LHSH__xhshlatch”，里面正式描述了该类型的问题，并且提供了三种解决方案，其中包括设置 DB2_APM_PERFORMANCE，这也是在手机银行问题中唯一有效的方式。

5. 案例总结

在这个案例中，面对紧急性能问题，第一原则是尽快恢复数据库服务水平。在简单查看数据库情况(例如 CPU、active session 等)之后，可以尝试重启数据库实例。如果时间允

许，在重启之前可以收集一些相关数据，例如 application snapshot、lock、latch 等。

如果重启数据库实例依然没有解决问题，则需要现场进行问题诊断，找到问题的根源，并有针对性地采取措施。例如在本案例中，通过查看 latch 发现是 latch 等待问题，进一步通过 stack 分析得出是 package lock 相关对象上 latch。尽管 latch 是数据库内部机制，无法直接控制其行为，但仍可以通过对其上层(package lock)的干预来避免 latch 的竞争。

另外两个经验教训是：一是要多学习和积累已有的问题案例，例如这次的问题 IBM 网站上之前已经给出了 technote，但我们没有及时采用；二是对于高并发的业务系统，要有限流机制，并且从应用程序层面到数据库层面都要有限流机制。需要注意的是，对于该案例，采用 DB2 WLM 限制并发数并不起作用。

11.8 调优学习案例：利用压力测试程序学习 DB2 调优

好了，到现在为止，您已经看完本书的所有内容，有没有想实际做一下性能调整呢？

在下面这个案例中，我们将对前面学习的内容做全面复习。因为在实际生活中，我们不可能对自己的生产环境进行调优，所以本章通过程序来模拟压力测试环境，并通过这个环境让读者进行调优练习，在这个练习中会用到我们在前面各个章节中学习的内容。希望大家能够做好模拟环境，仔细做好这个实验。

在这个案例中，我们将引导读者一步步地学习在监控和调优 DB2 数据库中对性能影响最大的配置参数，影响性能的维护操作(统计信息更新、索引创建等)也穿插在这个例子中。通过使用本书提供的示例 Java 程序——“DB2 性能测试程序”，您可以了解这些实用技术，而且还可以在您自己的系统上试验各种场景。在这个过程中，您可以使用 Java 程序来模拟对数据库执行 SQL 的工作负载。许多因素都可以影响数据库服务器的性能，但是这个实验将关注如何调优一些重要的 DB2 配置参数，以及用于捕获和修复“恶劣 SQL”的步骤。

1. 概述

性能是您使用数据库系统最重要的问题之一。有 3 个主要因素影响系统性能：CPU 使用、I/O 利用和内存使用，如图 11-15 所示。

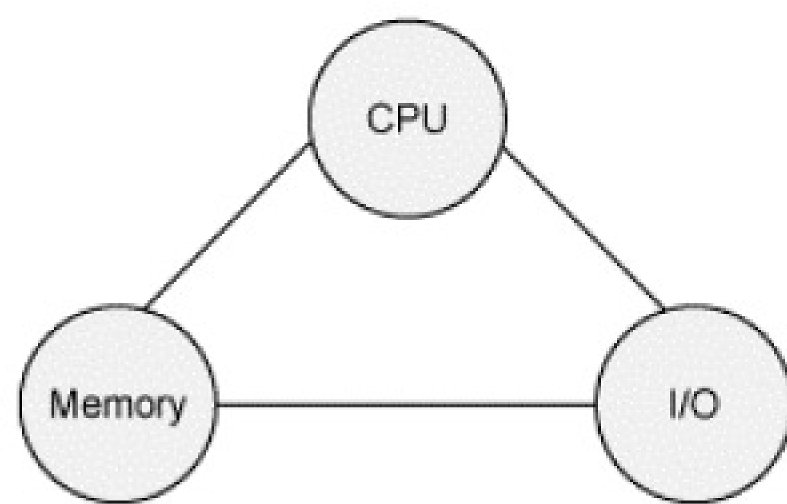


图 11-15　影响性能的主要因素

提示：

此处省去了网络，因为在实际生产中，由于网络引起的性能瓶颈很少。

在进行性能调优时，根据系统的具体情况，监控 CPU、内存和 I/O 是否存在性能瓶颈。

对数据库工作负载的理解对于有效配置数据库以获得最优性能也很关键。有以下几种类型的数据库工作负载：

- 联机事务处理(Online Transaction Processing，OLTP)，包含许多不同复杂性的小型事务。OLTP 事务包括 select、insert、update 和 delete，通常是以秒级或次秒级(sub-second)的速度完成的。
- 决策支持系统(Decision Support System，DSS)，通常是仅为 select 的事务，包含访问大量数据的大型查询。
- 批处理。
- 报表。
- 以上系统的混合。

无论数据库负载如何，通用的调优方法及流程都是相同的，只是 OLTP 和 DSS 工作负载之间存在差别。这个案例仅仅说明 OLTP 环境中的监控和调优提示。

许多因素都可以影响数据库服务器的性能，例如硬件系统的设计、数据库对象的设计、数据存储器的管理、应用程序的设计等等。这个实验将关注如何在一定的工作负载下调优 DB2 配置参数，以及用于捕获和修复“恶劣 SQL”的步骤。

2. “DB2 性能测试程序”和“TESTDB”数据库

我们提供了性能示例，这些示例使用名为 TESTDB 的数据库以及名为“DB2 性能测试程序”的简单的 Java 独立程序。创建和填充 TESTDB 数据库以及运行“DB2 性能测试程序”所需的文件和指示可以从笔者的博客下载。最多花费 5 分钟就可以准备好该环境。

出于简单性的考虑，TESTDB 数据库只包含两个表：ACCOUNT 和 AUDITLOG。“DB2 性能测试程序”是 JDBC 应用程序，一旦调用该程序，就使用 type 2 Universal 驱动程序并建立到 TESTDB 的 10 次连接。其接口极其简单。只要指定测试的持续时间并单击“运行”即可。当程序执行时，将在 10 次连接的每一次中用随机参数反复执行相同的事务。当调优数据库时，已完成的事务数目字段应随着调优的进行逐渐增加。这就是您的性能目标。

创建 TESTDB 数据库和 ACCOUNT、AUDITLOG 表的 SQL 如下：

```
create database testdb@
connect to testdb@
create sequence MYSEQ@
CREATE TABLE ACCOUNT
```

```
        (ACCT ID INT       NOT NULL,
         NAME         CHAR(20)  NOT NULL,
         BRANCH ID     INT   NOT NULL,
         BALANCE       INT       NOT NULL,
         ADDRESS       CHAR(30)  NOT NULL)@
begin atomic
   declare cnt INT default 0;
   while (cnt < 100000) do
      insert into account values (
         nextval for MYSEQ,
         'account' concat char(nextval for MYSEQ),
         ceiling((rand()*10)),
         ceiling((rand()*1000000)),
         'address' concat char(nextval for MYSEQ)
      );
 set cnt=cnt+1;
   end while;
end@
CREATE TABLE AUDITLOG
        (ACCT ID   INTEGER  NOT NULL,
         TELLER ID    INT   NOT NULL,
         BRANCH ID    INT   NOT NULL,
         BALANCE      INTEGER  NOT NULL,
         TRANSID      BIGINT  NOT NULL)@
drop sequence MYSEQ@
select count (*) from account@
select count (*) from auditlog@
```

把上述 SQL 存放到 testdb.sql 文件中，执行“db2 –td@ -vf testdb.sql”创建本实验所需要的表。

DB2 性能测试程序发出的事务包含下列查询：

- SELECT NAME，BALANCE FROM ACCOUNT WHERE ACCT_ID = <value>
- UPDATE ACCOUNT SET BALANCE = ? WHERE ACCT_ID = ?
- INSERT INTO AUDITLOG VALUES (?，?，?，?，?)

希望您边浏览内容边操作 DB2 性能测试程序(在测试环境中)。

3. 建立性能调整基线

在调优数据库之前，您通常应建立性能基线(baseline)。对于这个特定的场景，我们会跟踪应用运行时内存的消耗，以及未调优时数据库所完成的事务数目。首先从初始环境开始，执行下列操作：

(1) 关闭所有打开的窗口。

(2) 打开 DB2 命令窗口，然后发出命令：db2stop force。

(3) 打开 Windows 任务管理器，选择 Performance 选项卡。

(4) 在 DB2 命令窗口中，发出命令：db2start。同时，查看任务管理器中的内存使用是多少。启动实例要消耗一些内存，但是不多。

(5) 在 DB2 命令窗口中，发出命令：db2 list application。您应收到一条消息表明 Database System Monitor 未返回任何数据。

(6) 通过执行文件“DB2 性能测试程序.bat”启动“DB2 性能测试程序”，并查看任务管理器中的内存是如何增加的。正如前面提到的，“DB2 性能测试程序”一旦被调用，就建立 10 次连接。当对数据库建立第一次连接时，就分配数据库内存。此外，每次连接都与一个 DB2 代理程序相关，该代理程序也花费一些内存成本。

(7) 在 DB2 命令窗口中，发出命令：db2 list applications。现在，您将看到 java.exe 应用程序(DB2 性能测试程序)到 TESTDB 数据库的 10 次连接。

您的桌面将如图 11-16 所示。请保持该窗口布局。



图 11-16　“DB2 性能测试程序”及桌面布局

为了建立性能基线，运行“DB2 性能测试程序”10 秒(5 到 8 次)，并写下结果。您将注意到在没有任何调优时，所完成事务的数目最少，而且事务的数目将会随着数据库的调优逐步有所提高，但最终会达到极限。这是正常的。在第一次运行期间，不会将太多信息缓存到内存中；因此，第一次运行付出的 I/O 代价就是将一些数据页从硬盘读到内存中。

此时，我们已经获得了未调优数据库时的性能基线。您现在应获取完整的 DB2 快照；然而，我们不会在本节中讨论该主题，请读者参考“第 3 章：DB2 性能监控”。因此，这时您在阅读完本节之后需要从头开始重复该实验。

4. DB2 监控

对数据库系统的监控是调优数据库服务器的重要部分。监控数据的集合充当基线，可用于比较当前和过去的性能。这帮助我们更早更快地检测出性能问题。监控数据还有助于您理解数据库配置参数和应用程序修改对性能的影响，并帮助我们分析随着系统负载的增长性能趋势的发展情况。

1) 监控数据库服务器的技术概述

DB2 提供了多个工具，可以用于监控数据库服务器的活动，或分析 SQL 语句如何访问数据；每个工具都服务于不同的目的。表 11-5 列出了不同的监控工具。

表 11-5　DB2 监控工具概述(详细内容参见本书“第 3 章：DB2 性能监控”)

监 控 工 具	监 控 信 息
快照监控器	在特定时刻(获取快照的瞬间)及时地捕获数据库活动的状态信息
事件监控器	在发生诸如语句执行、事务完成，或者应用程序断开连接之类的特定数据事件时记录数据。STATEMENT 和 DEADLOCK 事件监控器常用于性能调优实践
解释工具	捕获关于 SQL 语句的访问计划和环境的信息，即如何执行单条 SQL 语句以访问数据
db2batch	监控 SQL 语句的性能特点和执行持续时间。从平面(flat)文件或标准输入中读取 SQL 语句，动态地准备和描述语句，并返回性能基准测试信息，如 SQL 语句的准备时间(Prepare Time)、执行时间(Execute Time)和读取时间(Fetch Time)等

本例中，我们使用快照监控器和 SQL 解释工具。

2) 快照监控器

快照监控器在不同的层次上收集信息，如表 11-6 所示。

表 11-6　快照监控器的层次

层　　次	所捕获的信息
数据库管理器	捕获活动数据库管理器实例的统计信息
数据库	为当前数据库分区上的所有活动数据库提供通用的统计信息
应用程序	提供连接到当前数据库分区上数据库的所有活动应用程序的有关信息

在每个数据库中，快照监控器基于表 11-7 所示的功能组层次收集信息。

表 11-7　快照的功能组层次

功能组层次	所捕获的信息
缓冲池活动	读写次数以及所花时间
锁	锁定的数目，死锁的次数
排序	所使用的堆的数目，溢出，排序性能
SQL 语句	启动时间，停止时间，语句标识
表活动	测量活动(行读，行写)
工作单元	开始时间，结束时间，完成状态

虽然默认情况下只为 表 11-7 所示的每个功能组层次收集一些基本信息，但也可以通过打开快照监控器开关在每个层次上收集更多详细的统计信息。因为监控包含了开销，所以您应该只打开在监控任务中最要紧的那个监控器开关。另一方面，如果您使用的是测试系统，我们就建议您在调优系统时打开所有的监控器开关。

通过分别使用 UPDATE DBM CFG 或 UPDATE MONITOR SWITCHES 命令，在实例或应用程序层打开或关闭监控器开关。当在应用程序层打开监控器开关时，如 DB2 命令窗口，该监控器就将仅仅应用于特定会话。例如，为了在应用程序层打开 BUFFERPOOL、SORT 和 STATEMENT 的监控器开关，就要从 DB2 命令窗口发出下列命令：

```
db2 update monitor switches using BUFFERPOOL ON SORT ON STATEMENT ON
```

因为您将在测试系统中使用 DB2 性能测试程序，所以出于简单性的考虑(不必跟踪哪个会话打开了监控器)，就通过从 DB2 命令窗口发出下列命令在实例层打开所有的监控器，如下所示(默认情况下，TIMESTAMP 和 HEALTH MONITOR 的开关为 ON)：

```
C:\> db2 update dbm cfg using DFT MON BUFPOOL   ON
C:\> db2 update dbm cfg using DFT MON LOCK      ON
C:\> db2 update dbm cfg using DFT MON SORT      ON
C:\> db2 update dbm cfg using DFT MON STMT      ON
C:\> db2 update dbm cfg using DFT MON TABLE     ON
C:\> db2 update dbm cfg using DFT MON TIMESTAMP ON
C:\> db2 update dbm cfg using DFT MON UOW       ON
C:\> db2 update dbm cfg using HEALTH_MON        ON
```

您或许还需要增加监控器的堆大小，因为这是用于存储快照监控器信息的内存区域。如果设置太小，信息会被快速覆盖掉。使用下列命令修改该参数：

```
db2 update dbm cfg using MON_HEAP_SZ 1024
```

上面的数据库管理器配置修改需要重新启动实例，以便修改生效。

为了找到当前的监控器开关设置，要发出：

```
db2 get monitor switches
```

该命令的输出如下所示：

```
               监视器记录开关
数据库分区号 0 的开关列表
缓冲池活动信息        (BUFFERPOOL) = ON  2008-10-24 09:20:37.560724
锁定信息                   (LOCK) = ON  2008-10-24 09:20:37.560724
排序信息                   (SORT) = ON  2008-10-24 09:20:37.560724
SQL 语句信息           (STATEMENT) = ON  2008-10-24 09:20:37.560724
表活动信息                (TABLE) = ON  2008-10-24 09:20:37.560724
获取时间戳记信息(时间戳记)    = ON  2008-10-24 09:20:37.560724
工作单元信息                (UOW) = ON  2008-10-24 09:20:37.560724
```

在打开合适的监控器开关之后，您可以在同一 DB2 命令窗口中使用 get snapshot 命令来收集监控统计信息。例如，尝试使用 TESTDB 数据库来获取快照。在 DB2 命令窗口中，发出下列命令：

```
C:\> db2 connect to TESTDB
C:\> db2 select * from sysibm.sysdummy1
C:\> db2 get snapshot for all on TESTDB
```

我们将在下面的内容中更详细地讨论快照监视。

3) SQL 解释工具(详细描述请参见本书“第 9 章：SQL 语句调优”)

SQL 解释工具提供查询优化器为 SQL 语句所选择的访问计划有关的详细信息。该信息存储在解释表中，可以在稍后使用诸如 Visual Explain、db2expln、dynexpln 和 db2exfmt 等工具进行格式化输出，从而以较友好的方式进行表示。

解释表可以在您第一次使用 Visual Explain 时自动进行创建。即使没有创建它们，您也可以手工进行创建，如下所示：

```
C:\> cd <db2 install path>\sqllib\misc
C:\> db2 connect to TESTDB
C:\> db2 -tvf EXPLAIN.DDL
```

本例中，我们使用 db2exfmt 工具。例如，使用 db2exfmt 解释动态 SQL 语句，在 DB2 命令窗口中按照下列步骤进行：

```
C:\> db2 connect to <database name>
C:\> db2 set current explain mode explain
C:\> db2 -tvf <Input file with an SQL statement ended with a semicolon>
C:\> db2 set current explain mode no
C:\> db2exfmt -d <dbname> -g TIC -w -1 -n C:\> -s C:\> -# 0 -o <output file>
```

db2exfmt 工具的输出包括表 11-8 所示信息。

表 11-8　db2exfmt 输出概述

区 域 名 称	内　　容
概述	DB2 版本和发布级别，以及运行解释工具时的日期和时刻
数据库环境	优化器为确定具有最少资源成本的访问计划所考虑的配置参数，包括并行度、CPU 速度、通信速度、缓冲池大小、排序堆大小、数据库堆大小、锁列表大小、最大锁列表、平均应用程序和可用锁
程序包环境	SQL 类型(动态的或静态的)、优化级别、隔离级别以及语句使用的分区内并行度
初始语句	应用程序调用的 SQL 语句
优化语句	优化器从初始语句进行转换的 SQL 语句的改写版本，这些语句具有相同查询结果，但允许最优性能
访问计划	允许 DB2 访问数据以解决 SQL 语句的最小扩展路径
操作符描述	展示访问计划的每个阶段(操作符)发生什么
访问计划中使用的对象	访问计划中使用的表和/或索引

我们将在下面的内容中提供更多关于 db2exfmt 工具的例子。

5. 调优 DB2 配置参数

DB2 数据库默认的配置参数值基于使用 256MB RAM 和单个磁盘的系统。如果您具有一个较大的系统，就需要修改这些参数，以便最好地利用系统资源。您可以通过使用 Configuration Advisor 确定调优配置的良好起点，Configuration Advisor 基于您的系统资源来推荐数据库参数值。为了运行 Configuration Advisor，要使用 autoconfigure 命令，或者通过 Control Center 调用它，右击想要配置的数据库，并选择 Configuration Advisor。或者使用数据库的自调优内存(self_tuning_mem)使数据库自动地调整部分内存的大小来适应负载的变化。

出于实验目的，我们不在本例中运行 Configuration Advisor，而是手工调优不同的参数。

这里描述的配置参数包括：

- 缓冲池
- 异步页面清理器和 I/O 服务器
- 排序
- 日志缓冲区大小
- 分组 COMMIT 语句

1) 缓冲池

在诸如读、写、更新和删除的所有事务性活动期间，缓冲池充当数据库执行大多数数据操作(除了大对象和长字段数据)在“内存中”的工作区域。每个数据库都至少需要一个缓冲池。对于具有大于一种页面大小的表空间的数据库而言，需要创建具有匹配页面大小的新的缓冲池。

缓冲池竞争可以成为影响数据库性能的重要因素。如果缓冲池大到足以在内存中保存所有需要的数据，就会发生最少的磁盘活动。相反，如果缓冲池太小，那么数据库的整体性能就可能受到严重影响，而在应用程序需要进行大量的磁盘活动的情况下，I/O 就很有可能成为整个数据库性能的瓶颈。

配置

为了确定数据库中缓冲池的大小，请查看如下语句：

```
C:\> db2 connect to TESTDB
C:\> db2 "select BPNAME,NPAGES,PAGESIZE from SYSCAT.BUFFERPOOLS"
BPNAME  NPAGES      PAGESIZE
-------------  -----------  -----------
IBMDEFAULTBP          250         4096
1 record(s) selected.
```

输出显示 TESTDB 数据库定义了一个缓冲池(IBMDEFAULTBP)，该缓冲池具有 250 个页面(NPAGES)，每个页面大小(PAGESIZE)为 4096 字节。Windows 上 IBMDEFAULTBP 的默认大小是 250 个页面；而在 UNIX 平台上，默认大小是 1000 个页面。当 NPAGES 的值为 –1 时，缓冲池大小就是由数据库配置参数中的 BUFFPAGE 参数确定的。例如：

```
C:\> db2 get db cfg for TESTDB
...
Buffer pool size (pages) (BUFFPAGE) = 250
```

监控

因为缓冲池的目的就是为了在内存中保存页面，用于数据库服务器所需的数据操作，

以免从磁盘读取页面，所以对于缓冲池有效性的一个重要测量方法就是看所请求的页面有多频繁地存在于缓冲池中。缓冲池命中率(hit ratio)测量该有效性，它可以按照下列方法进行计算：

```
    BPHR=(1- ((缓冲池数据物理读 + 缓冲池索引物理读)/(缓冲池数据逻辑读 + 缓冲池索引逻辑读)))
* 100%
```

命中率越接近 100%，磁盘 I/O 的频率就越低，因而读取数据的开销就越少。

监控缓冲池活动的常用方法是使用缓冲池快照，如下所示(注意：确保缓冲池监控器开关是 ON。否则，在快照监控期间，就不会收集大多数缓冲池统计信息)：

```
    C:\> db2 get snapshot for bufferpools on <database name>
    应该关注的缓冲池快照的重要元素如下：
    Buffer pool data logical reads = 16359
    Buffer pool data physical reads = 209
    Buffer pool index logical reads = 90
    Buffer pool index physical reads = 52
```

以上信息显示了 98.4%的良好缓冲池命中率。

实验

从命令提示符窗口启动 DB2 性能测试程序：

```
C:\> DB2 性能测试程序
```

图 11-17 显示了用一个包含 250 个 4KB 页面的缓冲池所完成的事务数目(注意：由于硬件资源不同，读者系统上的性能结果可能和此处的实验结果不同)。

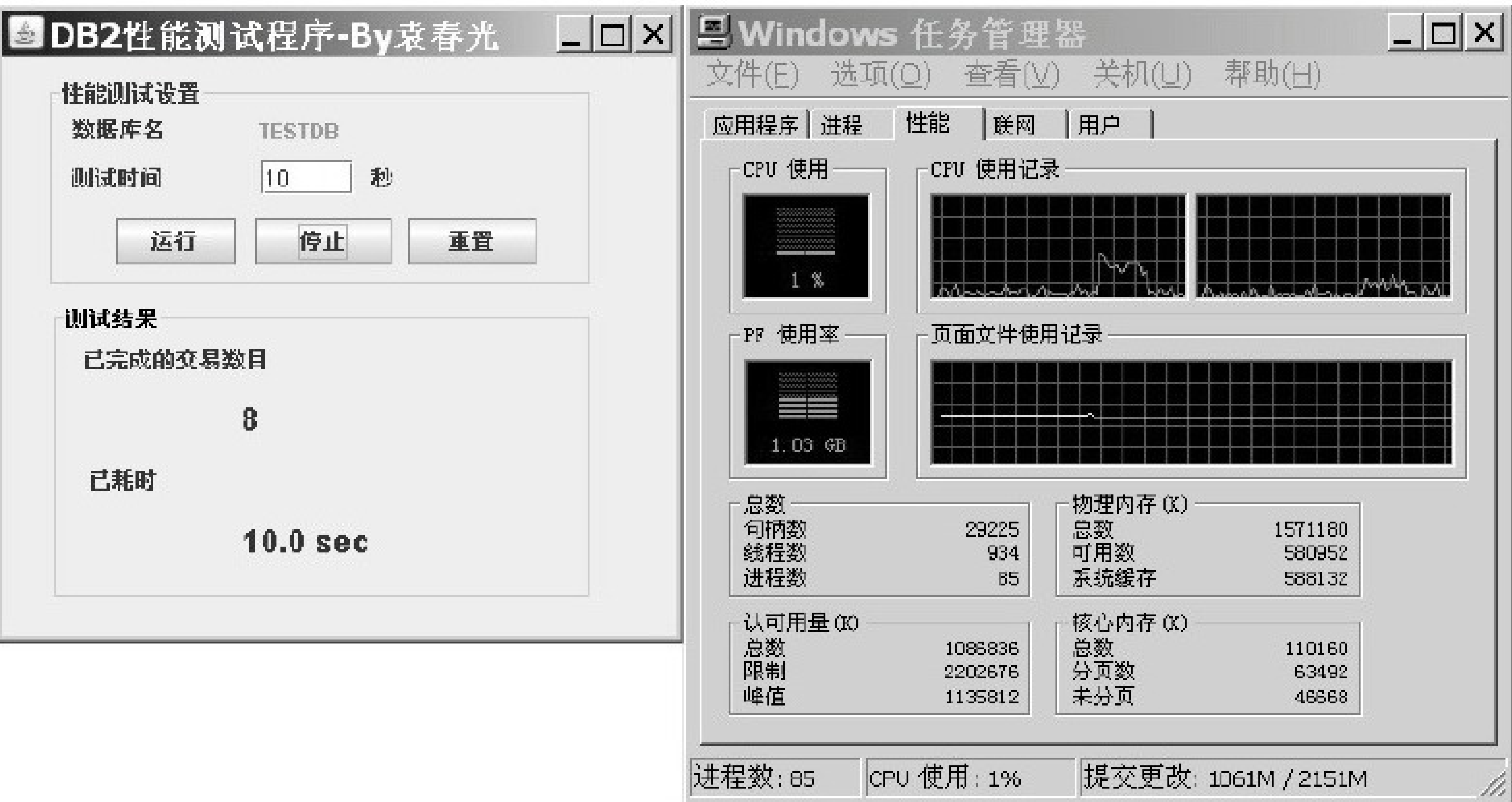


图 11-17　使用包含 250 个 4KB 内存页面的缓冲池的性能结果

在完成测试运行之后，就获取缓冲池快照，如下所示：

```
C:\> db2 get snapshot for bufferpools on TESTDB
…………………..
Buffer pool data logical reads = 75875
Buffer pool data physical reads = 59419
Buffer pool index logical reads = 296
Buffer pool index physical reads = 208
…………………
```

缓冲池命中率 =(1－(59419+ 208) / (75875+ 208)) * 100 = 21.63%，通过 250 个页面的缓冲池大小，应用程序测试运行呈现了 21.63%的糟糕的缓冲池命中率。

联机增加缓冲池大小，如下所示：

```
C:\> db2 connect to TESTDB
C:\> db2 "alter bufferpool IBMDEFAULTBP immediate size 12000"
```

在发出以上命令时，请查看任务管理器，以便看到内存使用是如何增加的。

监控信息是累积的，除非您重置监控器。因为我们需要在修改缓冲池大小之后获得新的缓冲池命中率，所以要在下一次测试运行之前用以下命令重新设置所有监控器：

```
C:\> db2 reset monitor all
```

单击“DB2 性能测试程序”屏幕上的“重置”按钮，然后单击“运行”。通过 12000 个 4KB 页面的更大缓冲池，性能结果从每 10 秒完成 8 个事务增加到完成 598 个事务，如图 11-18 所示。

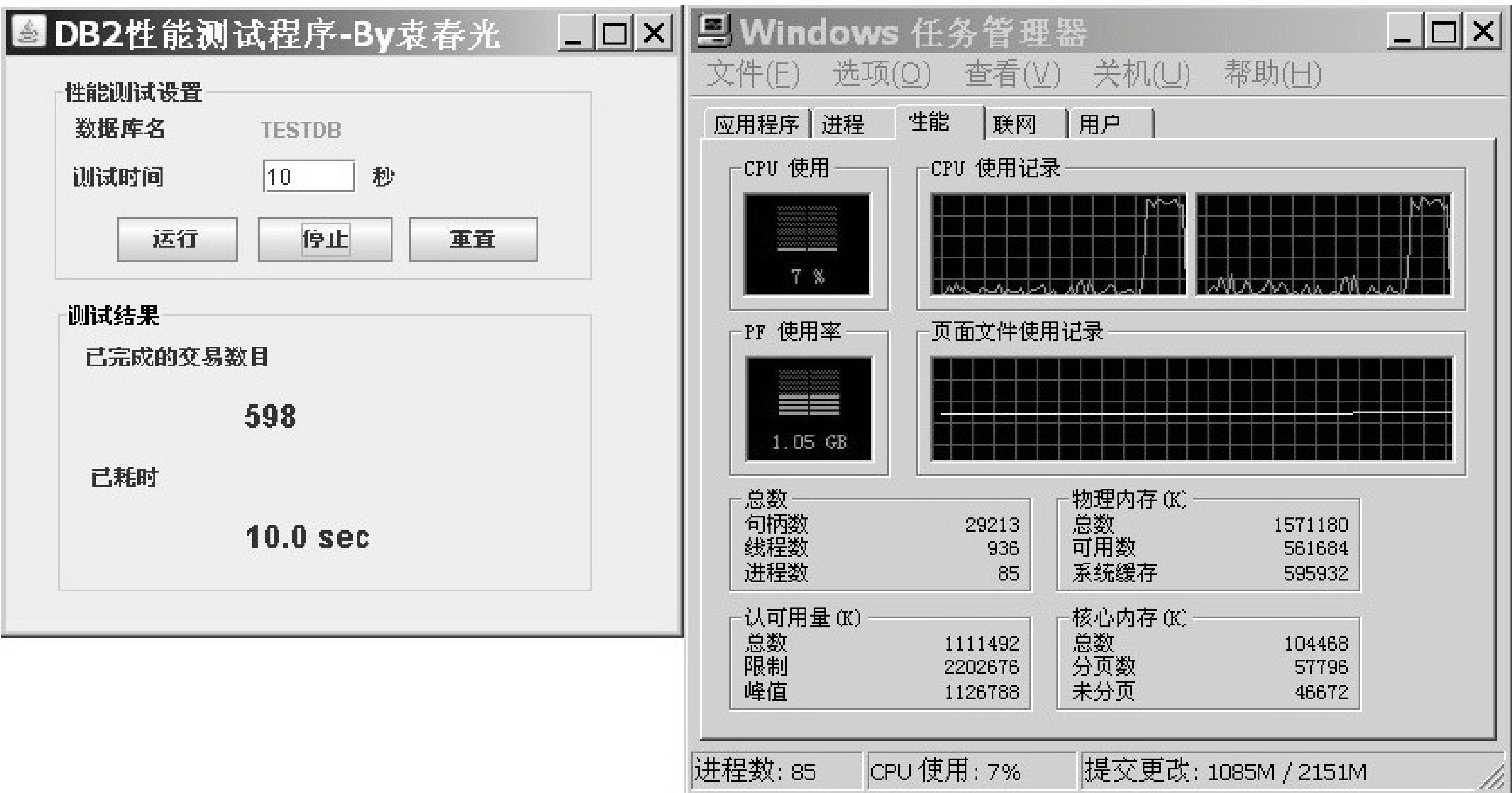


图 11-18　使用包含 12000 个 4KB 内存页面的缓冲池的性能结果

获取新的快照，并计算新的缓冲池命中率，如下所示：

```
C:\> db2 connect to TESTDB
C:\> db2 get snapshot for bufferpools on TESTDB
Buffer pool data logical reads = 2137439
Buffer pool data physical reads = 1594
Buffer pool index logical reads = 82
Buffer pool index physical reads = 50
```

缓冲池命中率 = (1 - (1594 + 50) / (2137439 + 82)) * 100% = 99.92%

现在的缓冲池命中率就显示了较好的结果。一般情况下，缓冲池命中率在 90%以上就认为是良好的。如果您的系统具有较低的缓冲池命中率，您就可以通过增加缓冲池大小来进一步取得更好的应用程序性能结果。如果将缓冲池大小增加到 20 000 个 4KB 页面并重新运行 DB2 性能测试程序，那将会发生什么情况呢？性能会有所提高吗？

在这种情况下，性能可能不会提高。TESTDB 数据库的大小约为 36 MB。这意味着整个数据库都可以进入内存。12000 个页面(48MB)或 20000 个页面(80MB)的大小将没有区别，因为整个数据库都可以包含在 48MB 中。实际上，取决于应用程序所执行的 SQL 的类型，将缓冲池的大小增加到某个极限之后就可能不会再提高性能了。我们建议您不断增加缓冲池大小，直到看到不再有性能提高为止。

与其他内存缓冲区相比，缓冲池对数据库性能的影响最为显著。但是请记住，缓冲池是操作系统共享内存集中的一部分。在 32 位的数据库环境中，DB2 具有数据库共享内存大小的限制，AIX 上是 1.75GB，Sun Solaris 上是 3.35GB，HP-UX 上是 0.75GB 到 1 GB 之间的值，Linux 上是 1.75GB，Windows 上是 2GB 或 3GB(假设 boot.ini 中启用了 3GB 开关)；因此，您需要平衡缓冲池与其他共享内存缓冲区的配置。在 64 位的数据库环境中，这是一个不需要考虑的问题。

2) 异步 I/O 服务器和页面清理器

DB2 鼓励缓冲池与磁盘之间页面读写的异步 I/O 访问，以便获得最优性能。

I/O 服务器(db2pfchr)将数据页异步地从磁盘读入参与应用程序需求的缓冲池中，这称作“预取”(prefetching)。预取可以提高数据库的性能，因为当代理访问这些页面时，将在缓冲池中找到它们，从而减少了应用程序等待页面从磁盘读入缓冲池的时间。

另一方面，在数据库代理需要缓冲池中的空间之前，页面清理器(db2pclnr)将已修改的页面从缓冲池写入磁盘。因此，数据库代理不必等待写出修改的页面，就可以使用缓冲池中的空间。这将提高数据库的整体性能。页面清理器可以通过多种理由触发，例如，当达到已修改页面的阈值时。异步 I/O 服务器、页面清理器和数据库其他组件的关系如图 11-19 所示。

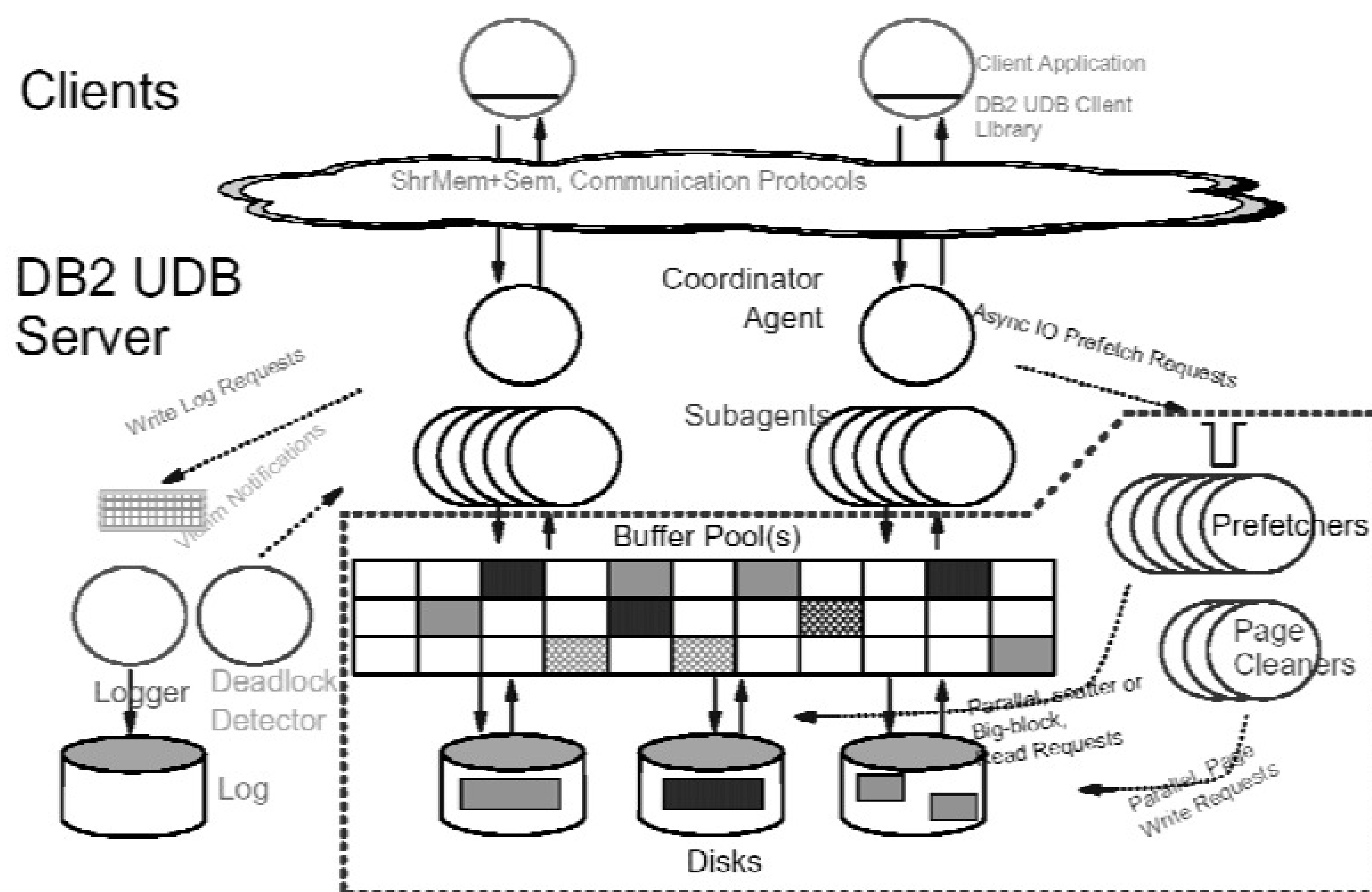


图 11-19　异步 I/O 服务器、页面清理器与数据库其他组件之间的关系

配置

用于数据库的 I/O 服务器(预取器)的数目可以使用 num_ioservers 数据库配置参数进行配置。为了充分利用系统中的所有 I/O 设备，使用较好的值，该值通常比数据库所驻留的物理设备的数目多一两个。最好配置更多的 I/O 服务器，因为每个 I/O 服务器相关的开销非常小，而未使用的 I/O 服务器仍然将保持空闲。

num_iocleaners 数据库配置参数允许您为数据库指定异步页面清理器的数目。在为该参数设置值时，要考虑下列因素：

- 应用程序类型：如果是一个不具有更新而仅仅查询的数据库，就将该参数设置为 0。当查询工作负载导致创建许多 TEMP 表(您可以通过使用解释实用程序来确定)时除外。如果事务是对数据库运行的，那么就将该参数设置成 1 到用于该数据库的物理存储设备数目之间的值。
- 工作负载：具有较高事务更新率的环境(OLTP)可能需要配置更多页面清理器。
- 缓冲池大小：具有大型缓冲池的环境也可能需要配置更多页面清理器。

还请记住，配置太多页面清理器可能会损害数据库服务器上的运行队列，并导致极大的性能下降。因此，根据经验，您可以考虑将页面清理器的数目设置为与数据库服务器上 CPU 的数目相同。

已修改页面的阈值参数(chngpgs_thresh)决定异步页面清理器应启动时已修改页面的百分比。

监控

因为异步页面清理器和异步 I/O 服务器的活动与缓冲池活动是如此紧密联系，所以您可以再次利用缓冲池快照来测量页面清理器和预取器的有效性。

这里，缓冲池快照的重要元素将重点放在了下面：

```
Buffer pool data logical reads = 2700145
Buffer pool data writes = 0
Buffer pool index logical reads = 95
Asynchronous pool data page reads = 85
Asynchronous pool data page writes = 0
Buffer pool index writes = 0
Asynchronous pool index page reads = 0
Asynchronous pool index page writes = 0
```

预取活动可以通过异步和同步的读 I/O 来确认。异步读取比率按照下列公式计算：

```
((异步池数据页读 + 异步池索引页读) / (缓冲池数据逻辑读 + 缓冲池索引逻辑读)) * 100%
```

异步读取比率值较小可能是由多种原因导致的，例如：

- 工作负载读写单行，因此无法利用预取。
- 为数据库配置的预取器太少。
- 数据库中的表空间仅仅配置了一个容器，以致无法进行预取。

异步页面清理器的有效性是由异步数据和索引页的写比率进行测量的。如果下列两个条件都成立，就可以减少异步页面清理器的数目(num_iocleaners)：

- 缓冲池数据写约等于异步池数据页写。
- 缓冲池索引写约等于异步池索引页写。

如果下列条件中有一个成立，就应增加该参数：

- 缓冲池数据写远远大于异步池数据页写。
- 缓冲池索引写远远大于异步池索引页写。

示例

首先，确认 num_ioservers 和 num_iocleaners 参数的当前设置：

```
C:\> db2 get db cfg for TESTDB
Number of asynchronous page cleaners (num iocleaners) = 1
Number of I/O servers (num_ioservers) = 3
```

在 DB2 性能测试程序的下一次运行之前，重置快照监控器：

```
C:\> db2 reset monitor all
```

这里，缓冲池快照通过应用前面讨论的规则，用于验证系统上是否正确配置了 num_ioservers 和 num_iocleaners 参数。

```
C:\> db2 get snapshot for bufferpools on TESTDB

Buffer pool data logical reads = 269482
Buffer pool data writes = 0
Buffer pool index logical reads = 82
Asynchronous pool data page reads = 1236
Asynchronous pool data page writes = 0
Buffer pool index writes = 0
Asynchronous pool index page reads = 0
Asynchronous pool index page writes = 0
```

异步读取比率 = ((1236 + 0) / (269482 + 82)) * 100% = 0.004%

异步读取比率如此小的原因是数据库中的每个表空间仅仅设置了一个容器，以致无法进行预取。因此，num_ioservers 参数的当前设置可以保持相同。

然后，您需要检查缓冲池数据写(0)与异步池数据页写(0)以及缓冲池索引写(0)与异步池索引写(0)的监控数据。通过应用前面讨论的规则，表明 num_iocleaners 参数不需要进行进一步的调优工作。

chngpgs_thresh 的 60%的默认值对于 OLTP 工作负载而言通常太高了。介于 20%和 40%之间的值会更好一些。例如，如果您具有一个 2GB 的缓冲池，当达到 60%的修改页面时，在触发页面清理器时就有 1.2GB(2GB 的 60%)写入磁盘。当这发生时，可能导致系统整体速度的下降。通过将 chngpgs_thresh 设置为较低的数，如 20%，将更频繁地触发页面清理器，但只有较少的数据写入磁盘，而用户也觉察不到这一延迟。

3) 排序

DB2 具有两种基本类型的排序：共享排序和私有排序。

只有 intra_parallel 数据库管理器配置参数是 ON 或启用集中器(concentrator)时(当 max_connections 大于 max_coordagents 时)，才可以使用共享排序。通常在需要具有多个子代理供给(subagent feeding)或从排序中进行预取时使用它们。用于共享排序的内存是从数据库共享内存集中分配的。

当 intra_parallel 参数是 off，并且禁用集中器(concentrator)时，所有的排序就都是私有的。装入和创建索引的操作通常使用私有排序进行索引键的排序，无论 intra_ parallel 参数的值如何。用于私有排序的内存是从代理的私有内存中分配的。因此，私有排序只能被单个代理访问。

对于共享排序，sheapthres_shr 数据库配置参数是数据库范围的可以用于任何一次排序

的数据库共享内存总数的硬限制。当用于活动共享排序的共享内存总数达到了该极限时，随后的排序就会以 SQL0955 错误码失败。如果 sheapthres_shr 的值是 0，那么共享排序内存的阈值将等于数据库管理器配置参数 sheapthres 的值，该参数也用于限制 DB2 实例中可用的私有排序内存总和的最大值。

在共享排序和私有排序中，sortheap 数据库配置参数是将用于单个排序的 4KB 内存页面的最大数目。

同样，排序包含两个阶段：

- 排序阶段：排序可以是溢出的，也可以是非溢出的。如果无法将排序的数据整个放入排序堆(sorhteap)中(排序堆是每次执行排序时分配的一块私有内存)，就会溢出到数据库的临时表中。显然由于不溢出的排序是在内存中完成，因此通常执行得比那些溢出的排序效率要高。
- 排序结果的返回阶段：返回可以是管道的(piped)，也可以是非管道的(non-piped)。如果排序信息可以直接返回，而无需临时表来存储最终排序的数据列表，那就是管道排序(piped sort)。如果排序信息的返回需要临时表，那就是非管道排序(non-piped sort)。所以管道排序的效率通常比非管道排序的效率要高。

配置

配置 sheapthres：

- 通常应将 sheapthres 设置为 sortheap 的倍数。
- 通常的经验是将 sheapthres 至少设置为 10 * sortheap。
- 对于带有 intra_parallel on 的索引创建，请确保 sheapthres >= d * sortheap，其中 d 是 SMP 度(degree)。
- 对于 LOAD，所有索引键的排序都是在进程的私有内存空间中同时进行的：SMP 中的 db2lrid，串行中的 db2lfrm0。要确保 sheapthres >= n * sortheap，其中 n 是装入的表上的索引数目。还请记住，AIX 上有最大为 250MB 的私有虚拟内存用于 LOAD 中的索引键排序(这是内存使用的限制)。

配置 sortheap：

- 增加 sortheap 的值可以显著地提高排序性能，因为能够在内存中完成的排序将更多，而相应的磁盘 I/O 将减少。
- 将 sortheap 配置小了可能导致由于溢出而引起的 I/O 增加所带来的性能下降。
- 将 sortheap 配置大了则导致容量问题，因为将会更快地超出 sheapthres 的限制。同样，因为到达 sheapthres_shr(用于共享排序)或超出 sheapthres(用于私有排序)，分配给新排序的内存数目将不断减少，这可能也会导致性能下降。

监控

除了缓冲池，排序是影响数据库性能的另一重要因素。DB2 监控排序活动的很多方面。数据库管理器快照展示下列监控元素：

```
C:\> db2 get snapshot for database manager
Private Sort heap allocated = 0
Private Sort heap high water mark = 80
Post threshold sorts = 0
Piped sorts requested = 167
Piped sorts accepted = 167
```

数据库快照展示下列监控元素：

```
C:\> db2 get snapshot for database on <database name>
Total Private Sort heap allocated = 0
Total Shared Sort heap allocated = 0
Shared Sort heap high water mark = 0
Total sorts = 170
Total sort time (ms) = 5015
Sort overflows = 93
Active sorts = 0
Commit statements attempted = 446
Rollback statements attempted = 24
```

下面列出了几个用于评估排序活动性能的关键公式：

阈值后排序(Post threshold sort)是指在超出排序堆阈值之后，请求堆的排序数目。在到达排序堆阈值之后开始的排序可能不会得到适量的内存去执行排序。

阈值后排序率 =(阈值后排序 / 总排序) * 100%

如果阈值后排序率很高，您就应该增加排序堆阈值(sheapthres)，或通过修改 SQL 查询来调整应用程序以使用较少或较小的排序内存。

管道排序率 =(接收的管道排序 / 请求的管道排序) * 100%

如果管道排序率的值很低，您就应该考虑增加排序堆阈值(sheapthres)以获得更高的排序性能。

排序溢出率 =(排序溢出 / 总排序) * 100%

如果排序溢出率的值很高，您就应该增加排序堆(SORTHEAP)，并且/或者也增加排序堆阈值(sheapthres)。

每个事务的排序 = 总排序 /(尝试的提交语句 + 尝试的回滚语句)

当每个事务有 3 个或更多排序时，就不应该首先调优排序堆或排序堆阈值，而是应该首先查找应用中的问题源，通过动态 SQL 快照确定执行较差的 SQL 语句，并在必要处添

加合适的索引。

示例

DB2 性能测试程序是 OLTP 应用程序，因此没有复杂的 SELECT 查询，排序也很少。这里的示例用于检查读者自己环境中的排序调优技术。

这里的准备步骤是为了将 ibmdefaultbp 的大小重新减小到 250 个页面，将 sortheap 减小到 16 个页面，以便您可以在小型数据库环境中查看相关的性能调优技术。

```
C:\> db2 connect to TESTDB
C:\> db2 "alter bufferpool ibmdefaultbp size 250"
C:\> db2 update db cfg for TESTDB using sortheap 16
C:\> db2 force applications all
C:\> db2 connect to TESTDB
```

为了重置快照监控器，要发出：

```
C:\> db2 reset monitor all
```

您可以创建 SQL DDL 文件 order_by.ddl，包含该 SQL：

```
SELECT NAME,BALANCE FROM ACCOUNT ORDER BY BALANCE
```

为了在数据库中触发排序活动，要重复下列命令多次，例如 3 次：

```
C:\> db2 -tvf order_by.ddl
```

数据库管理器快照可以用于确定阈值后排序活动：

```
C:\> db2 get snapshot for database manager
...
Private Sort heap allocated = 0
Private Sort heap high water mark = 256
Post threshold sorts = 0
Piped sorts requested = 3
Piped sorts accepted = 3
```

其中没有阈值后排序活动。然而，管道排序是 100%接受的，管道排序率 = (3 / 3) * 100% = 100%。

然后，可以利用数据库快照来确定数据库层的排序活动：

```
C:\> db2 get snapshot for database on TESTDB
...
Total Private Sort heap allocated = 0
Total Shared Sort heap allocated = 0
```

```
Shared Sort heap high water mark = 0
Total sorts = 3
Total sort time (ms) = 1097
Sort overflows = 3
Active sorts = 0
Commit statements attempted = 4
Rollback statements attempted = 5
```

排序溢出率 = (3 / 3) * 100% = 100%。由于这个较高的排序溢出率，您需要增加排序堆(sortheap)和/或排序堆阈值(sheapthres)。

按照通过快照监控进行的排序活动的健康检查，您现在需要增加排序堆(SORTHEAP)和/或排序堆阈值(sheapthres)。

为了增加 sortheap 参数，要发出：

```
C:\> db2 update db cfg for TESTDB using sortheap 400
C:\> db2 force applications all
```

在每次修改中，使用快照监控来确定是否需要进一步的排序调优。如果需要，就重复相同的步骤。

4) 日志缓冲区大小

日志缓冲区作为内存中的分级区域(staging area)来保存日志记录，而不是让 DB2 引擎直接将每条日志记录写入磁盘中。

当发生下列条件之一时，将日志记录写入磁盘：

- 达到 mincommit 配置参数所定义的事务提交或事物组提交
- 日志缓冲区已满
- 发生了一些其他的内部数据库管理器事件

配置

日志缓冲区大小是由 logbufsz 数据库参数定义的。如果在某个专用的日志磁盘上存在大量的读活动，或者具有较高的磁盘利用率，就应该增加日志缓冲区的大小。在增加 LOGBUFSZ 参数的值时，您还应该考虑数据库堆(dbheap)参数，因为日志缓冲区的内存来自数据库堆中的空间。

监控

可以通过查看下列快照元素，使用数据库快照来确定 logbufsz 数据库参数是否最优：

```
C:\> db2 get snapshot for database on <database name>
Log pages read = 0
```

```
Log pages written = 6721
```

日志页面读(log pages read)是日志记录器(logger)从磁盘读取的日志页面的数目，而日志页面写*(log pages written)*是日志记录器(logger)写入磁盘的日志页面的数目。日志页面读数目与日志页面写数目的比值应尽可能小。理想情况下，应该没有日志页面读。如果看到较多数目的日志页面读，就表示应该增加 logbufsz 数据库参数的值。

示例

对于具有许多更新工作负载的数据库系统而言，日志缓冲区的默认大小(logbufsz)通常都太小：

```
C:\> db2 get db cfg for TESTDB
Log buffer size (4KB) (LOGBUFSZ) = 8
```

DB2 性能测试程序没有太多的数据更新。因此，您不会在这里的 DB2 性能测试程序的快照中发现 logbufsz 参数需要调优的提示。但是，您仍然可以在自己的应用程序中进行尝试。如果您的应用程序碰巧具有比较高的数据更新工作负载，那么数据库快照监控就可以暴露大量的日志页面读。如果情况如此，您就可以考虑增加 logbufsz 数据库参数。通常应将它增加为不少于 256 页面。

5) 组中提交数目

在发生许多较短并发事务的环境中，每条 COMMIT 语句默认触发对磁盘的一次日志缓冲区刷新(flush)。因此，日志记录器(logger)进程频繁地将少量日志数据写入磁盘中，这就会导致大量的日志 I/O 开销而引起的性能下降。提交分组(commit grouping)允许到达请求提交数目的最小值之后才将日志缓冲区数据写入磁盘。该功能可以通过减少日志 I/O 的数量来提高性能。

仅仅当 mincommit 参数的值大于 1 且连接到数据库的应用程序数目大于或等于该参数的值时，才发生这种提交分组。在执行提交分组时，可以将应用程序的提交请求挂起，直到 1 秒钟之后或提交请求的数目等于 mincommit 参数的值。

配置

如果多个读/写应用程序通常请求并发的数据库提交，就从默认值开始增加 mincommit 参数。这将导致更高效的日志文件 I/O，因为它将发生得不那么频繁，并在每次发生时写入更多日志记录。

也可以对每秒的事务数目进行取样，并调整该参数以适应每秒事务数目的峰值(或它的某些较大比例)。适应峰值活动将最小化事务密集期间日志记录的写开销。

监控

数据库快照监控器可以用于确定每秒执行的事务数目，如下所示：

```
C:\> db2 get snapshot for database on <database_name>

Last reset timestamp = 07/30/2005 15:54:22.392292
Snapshot timestamp = 07/30/2005 19:24:10.858723
Commit statements attempted = 13784
Rollback statements attempted = 134
```

每秒的事务数目：

((尝试的提交语句 + 尝试的回滚语句) / (快照时间戳 - 最后重置的时间戳))

示例

MINCOMMIT 参数可以给数据库性能带来正面或负面的影响，所以需要正确设置。如果是在低并发的环境设置该参数，那么可能会给你的系统带来负面的性能影响。

```
C:\> db2 get snapshot for database on TESTDB

Last reset timestamp = 07/30/2005 19:32:45.570089
Snapshot timestamp = 07/30/2005 19:33:14.650596
Commit statements attempted = 7374
Rollback statements attempted = 1
```

每秒的事务数目 =(7374 + 1) / (19:33:14.650596 - 19:32:45.570089) = 254.13。

考虑到 DB2 性能测试程序仅仅调用较少的 10 个并发应用程序连接，而事务的数目很大(每秒 254 个事务)，所以不应修改 mincommit 参数的默认值；否则，将显著增加那些小事务。您可以自由地选择一些设置进行测试，以观察各种情况下的效果。

6. 数据库优化器

DB2 中包含了功能强大的基于成本评估的查询优化器，它用于确定访问数据的最佳策略。DB2 查询优化器总是试图通过将初始查询改写成优化的形式，生成备选查询执行计划，对每种备选计划的 I/O、CPU、内存和通信使用进行建模，并选择成本最小的访问计划执行，这样就能确定对数据库执行访问的效率最高的方法。

1) 更新目录统计信息

查询优化器使用数据库中的 SYSSTAT 目录视图来检索数据库对象的统计信息，并确定访问数据库的最佳方法。如果无法获得当前准确的统计数据，优化器就可能会基于不准确的默认统计数据选择低效的访问计划。

默认情况下，对于新创建的数据库，直到执行了 RUNSTATS 命令之后，才会收集数据库对象统计数据并将之存储在 SYSSTAT 目录视图中。如果还没有填充目录统计数据，那么 SYSSTAT 目录视图中诸如 CARD、NPAGES、FPAGES 等列就具有值 - 1。下面展示了一个示例。

确定是否运行了 RUNSTATS：

```
C:\> db2 connect to TESTDB
C:\> db2 describe table sysstat.tables
C:\> db2 "select tabname,card,npages,fpages from sysstat.tables"
TABNAME       CARD      NPAGES      FPAGES
------------ -------- ---------- ----------
ACCOUNT            -1         -1         -1
AUDITLOG           -1         -1         -1
```

我们强烈建议您使用 RUNSTATS 命令收集表和索引上的当前统计数据，特别是当发生了大量的更新活动或由于上次执行 RUNSTATS 命令而创建了新的索引时。这就为优化器提供了用于确定最佳访问计划的最准确的信息。例如：

```
    C:\> db2 runstats on table <table name> with distribution and detailed
indexes all
```

其中，<table_name>是全限定的表名，其中包含了模式名。

有时候，可能需要对数据库中的所有表执行 RUNSTATS 操作。这样做最容易的方法就是使用 REORGCHK 命令：

```
C:\> db2 reorgchk update statistics on table all
```

如果对于确定数据库是否包含表和索引上的最新统计数据有困难，您就可以发出下列命令来验证执行 RUNSTATS 操作的最近时间：

```
C:\> db2 "select name, stats_time from sysibm.systables"
```

如果仍然还没有运行 RUNSTATS，您将会看到 *stats_time* 列为"-"。否则，它就返回最后一次运行 RUNSTATS 的时间戳。

在下面的内容中，我们将在运行 RUNSTATS 之后测试 DB2 性能测试程序。

2) 缓存糟糕的查询

应用程序可以运行上百条不同的 SQL 语句；如果其中只有一条的代码不正确或未调至最优，那就可能会影响整个系统的性能。您如何才能捕捉这些糟糕的查询呢？

动态 SQL 语句快照提供了应用程序所运行的动态 SQL 语句的有关信息。按照这些步

骤重置监控器，运行 DB2 性能测试程序，然后获得动态 SQL 语句的快照：

```
C:\> db2 reset monitor all
C:\> 运行 DB2 测试程序
C:\> db2 get snapshot for dynamic sql on TESTDB
```

下面展示了部分动态 SQL 快照的输出。

动态 SQL 快照：

```
Dynamic SQL Snapshot Result
Database name                          = TESTDB
Database path                          = C:\DB2\NODE0000\SQL00006\
Number of executions                    = 1
Number of compilations                  = 1
Worst preparation time (ms)              = 0
Best preparation time (ms)               = 0
Internal rows deleted                   = 0
Internal rows inserted                  = 0
Rows read                              = 100000
Internal rows updated                   = 0
Rows written                           = 0
Statement sorts                         = 0
Statement sort overflows                 = 0
Total sort time                         = 0
Buffer pool data logical reads            = 1820
Buffer pool data physical reads           = 1727
Buffer pool temporary data logical reads   = 0
Buffer pool temporary data physical reads  = 0
Buffer pool index logical reads           = 0
Buffer pool index physical reads          = 0
Buffer pool temporary index logical reads  = 0
Buffer pool temporary index physical reads = 0
Total execution time (sec.ms)             = 0.493177
Total user cpu time (sec.ms)              = 0.040057
Total system cpu time (sec.ms)            = 0.010014
Statement text                          = SELECT NAME, BALANCE FROM
                                       ACCOUNT WHERE ACCT_ID =14680

Number of executions                    = 1
Number of compilations                  = 1
Worst preparation time (ms)              = 0
Best preparation time (ms)               = 0
Internal rows deleted                   = 0
Internal rows inserted                  = 0
```

```
   Rows read                              = 100000
   Internal rows updated                  = 0
   Rows written                           = 0
   Statement sorts                        = 0
   Statement sort overflows               = 0
   Total sort time                        = 0
   Buffer pool data logical reads         = 1820
   Buffer pool data physical reads        = 1527
   Buffer pool temporary data logical reads   = 0
   Buffer pool temporary data physical reads  = 0
   Buffer pool index logical reads        = 0
   Buffer pool index physical reads       = 0
   Buffer pool temporary index logical reads  = 0
   Buffer pool temporary index physical reads = 0
   Total execution time (sec.ms)          = 1.034426
   Total user cpu time (sec.ms)           = 0.040058
   Total system cpu time (sec.ms)         = 0.000000
   Statement text                         = SELECT NAME, BALANCE FROM
                                           ACCOUNT WHERE ACCT ID =47030

   Number of executions                   = 73
   Number of compilations                 = 1
   Worst preparation time (ms)            = 132
   Best preparation time (ms)             = 132
   Internal rows deleted                  = 0
   Internal rows inserted                 = 0
   Rows read                              = 0
   Internal rows updated                  = 0
   Rows written                           = 73
   Statement sorts                        = 0
   Statement sort overflows               = 0
   Total sort time                        = 0
   Buffer pool data logical reads         = 78
   Buffer pool data physical reads        = 28
   Buffer pool temporary data logical reads   = 0
   Buffer pool temporary data physical reads  = 0
   Buffer pool index logical reads        = 0
   Buffer pool index physical reads       = 0
   Buffer pool temporary index logical reads  = 0
   Buffer pool temporary index physical reads = 0
   Total execution time (sec.ms)          = 1.383423
   Total user cpu time (sec.ms)           = 0.000000
   Total system cpu time (sec.ms)         = 0.000000
```

```
    Statement text                          = INSERT INTO
                                             AUDITLOG VALUES (?,?,?,?,?)
     ...
```

我们提取了与 3 个不同查询相对应的 3 部分动态 SQL 快照：

```
    SELECT NAME, BALANCE FROM ACCOUNT WHERE ACCT ID =14680
    SELECT NAME, BALANCE FROM ACCOUNT WHERE ACCT ID =47030
    INSERT INTO AUDITLOG VALUES (?,?,?,?,?)
```

请注意，除了 ACCT_ID 的值之外，这两条 SELECT 语句完全相同。本例中故意将 DB2 性能测试程序编码为不具有参数标记(?)，以显示您不应做什么！DB2 优化器将这两条 SELECT 语句看作不同的查询，因此将分别编译每条查询从而导致额外开销，如以下字段所示：

```
    Number of executions = 1
    Number of compilations = 1
```

另一方面，如果查看 INSERT 语句，就可以看到它使用了 5 个参数标记(？)字段：

```
    Number of executions = 73
    Number of compilations = 1
```

上面显示只需要一次编译，具有不同值的同一查询执行了 73 次。

接着，让我们捕获糟糕的 SQL 语句。通过操作系统命令，您可以将快照命令的输出重新定向到文件，如下所示：

```
    C:\> db2 get snapshot for dynamic sql on TESTDB > snap1.txt
```

然后，可以使用编辑器查找字段“Total execution time”(如果是中文环境，查找“执行数”，其他字段类似)。按照降序对结果进行排序，并开始分析具有最高值的查询。例如，在我们的 Windows 系统中，我们安装了一个 UNIX 模拟器；因此，我们可以使用 grep 命令，如下所示：

```
    C:\> grep -i 'Total execution' snap1.txt
```

这将提供一个列表，如下所示：

查找糟糕的查询

```
    Total execution time (sec.ms)      = 0.471460
    Total execution time (sec.ms)      = 1.034426
    Total execution time (sec.ms)      = 0.354776
    Total execution time (sec.ms)      = 0.197684
```

```
Total execution time (sec.ms)     = 0.401673
Total execution time (sec.ms)     = 0.820391
Total execution time (sec.ms)     = 0.000000
Total execution time (sec.ms)     = 0.532227
Total execution time (sec.ms)     = 0.436514
Total execution time (sec.ms)     = 0.445380
Total execution time (sec.ms)     = 0.943996
Total execution time (sec.ms)     = 0.300309
Total execution time (sec.ms)     = 41.844554
Total execution time (sec.ms)     = 1.058051
Total execution time (sec.ms)     = 0.394608
...
```

虽然我们没有排序该列表，但可以立即看到具有最高执行时间的查询是 41.844554。下面展示了动态 SQL 快照输出中针对该查询的那部分。

糟糕查询的动态 SQL 快照区域

```
Number of executions                = 77
Number of compilations               = 1
Worst preparation time (ms)          = 0
Best preparation time (ms)           = 0
Internal rows deleted               = 0
Internal rows inserted              = 0
Rows read                        = 7700000
Internal rows updated               = 0
Rows written                      = 77
Statement sorts                    = 0
Statement sort overflows             = 0
Total sort time                    = 0
Buffer pool data logical reads      = 140140
Buffer pool data physical reads     = 124493
Buffer pool temporary data logical reads   = 0
Buffer pool temporary data physical reads  = 0
Buffer pool index logical reads     = 0
Buffer pool index physical reads    = 0
Buffer pool temporary index logical reads  = 0
Buffer pool temporary index physical reads = 0
Total execution time (sec.ms)      = 41.844554
Total user cpu time (sec.ms)       = 2.743941
Total system cpu time (sec.ms)     = 1.061525
Statement text                     = UPDATE ACCOUNT SET BALANCE = ?
                                        WHERE ACCT_ID = ?
```

通过分析该输出，我们注意到该查询已经执行了 77 次。因此，41.844554 的总执行就

是 77 次执行。用 41.844554 除以 77，我们得到每个查询的执行就是 0.5434358。因此，它并非和看上去的一样糟糕。是否可以提高该查询呢？当然可以。行读取的数目(77)表明很可能发生表扫描。然而，这并非一定是代价最高的查询。如果我们重复与前面相同的过程，就会发现具有该执行时间的查询：

```
Total execution time (sec.ms) = 1.034426
```

实际上是花费最多时间的查询。有问题的查询是：

```
SELECT NAME, BALANCE FROM ACCOUNT WHERE ACCT_ID =47030 s
```

快照中针对该查询的完整部分已经包含在前面的例子中了。下面我们讲解如何调整该查询。

3) 理解访问计划

此处关注如何理解访问计划本身，以便您可以确定糟糕查询的根本原因，然后解决该问题。

下面的示例通过分析我们在前面捕获的糟糕查询，介绍访问计划的基础知识。

创建脚本文件 *select.ddl*，其中包含下列 SELECT 语句：

```
SELECT NAME, BALANCE from ACCOUNT WHERE ACCT_ID=47030;
```

可以执行下列步骤，用于为以上 SELECT 语句生成访问计划：

```
C:\> db2 connect to TESTDB
C:\> db2 set current explain mode explain
C:\> db2 -tvf select.ddl
C:\> db2 set current explain mode no
C:\> db2exfmt -d TESTDB -g TIC -w -1 -n C:\> -s C:\> -# 0 -o explain.out
```

db2exfmt 输出的结构

在 *explain.out* 文件中，您可以看到 db2exfmt 的输出显示了许多关于数据库环境和查询的宝贵信息。下面列出了一些重要的区域。

Database Context 区域列出了优化器在确定具有最少资源成本的访问计划时所考虑的配置参数。

db2exfmt 输出的 Database Context 区域：

```
Database Context:
----------------
      Parallelism:          None
      CPU Speed:            9.053265e-007
```

```
          Comm Speed:             0
          Buffer Pool size:        250
          Sort Heap size:         400
          Database Heap size:      600
          Lock List size:         50
          Maximum Lock List:       22
          Average Applications:    1
          Locks Available:        1122
```

Package Context 区域提供了程序包的细节，例如 SQL 是动态的还是静态的、优化级别以及隔离级别。更重要的是，您可以看到 section 编号以及从何处发出的查询(QUERYTAG: CLP)。当您将访问计划重新匹配事件监控器输出时，这极其有用。它允许我们更好地跟踪特定的查询和有关事件。

db2exfmt 输出的 Package Context 区域：

```
Package Context:
---------------
        SQL Type:              Dynamic
        Optimization Level:      5
        Blocking:              Block All Cursors
        Isolation Level:        Cursor Stability

---------------- STATEMENT 1  SECTION 201 ----------------
        QUERYNO:               4
        QUERYTAG:              CLP
        Statement Type:         Select
        Updatable:             No
        Deletable:             No
        Query Degree:           1
```

下面展示了初始查询语句及其优化版本。

db2exfmt 输出的初始和优化区域：

```
Original Statement:
------------------
select name,balance  from account   where acct id=47030
Optimized Statement:
-------------------
SELECT Q1.NAME AS "NAME", Q1.BALANCE AS "BALANCE"
FROM YUAN.ACCOUNT AS Q1
WHERE (Q1.ACCT_ID = 47030)
```

下面展示了 SELECT 语句的访问计划。

db2exfmt 输出的 Access Plan 区域：

```
Access Plan:
-----------
        Total Cost:            3420.41
        Query Degree:           1
      Rows
     RETURN
     (   1)
      Cost
       I/O
       |
        1
     TBSCAN
     (   2)
     3420.41
      1820
       |
     100000
 TABLE: YUAN
     ACCOUNT
```

其余的两个区域提供了关于访问计划的每个操作符以及查询所使用的表和/或索引的细节。

访问计划操作符

下面展示了访问计划的基本组件：

```
    cardinality
<access plan operator>
      ( # )
       cost
     I/O cost
```

基数(cardinality)表示访问计划操作符返回的行的估计数目。访问计划操作符要么是必须在数据上执行的动作，要么是表或索引的输出。成本(cost)表示该操作以及前面操作的 CPU 累计成本，而 I/O 成本(I/O cost)表示 I/O 子系统中的操作符成本。

成本单位(unit of cost)是 *timeron*。*timeron* 不直接等于真正的时间消耗，但是对数据库管理器所需的资源(成本)给出了相对粗略的估算。

下面是您可以在本书中看到的一些访问计划操作符例子。

- RETURN：表示从查询到用户的数据返回。
- FETCH：使用指定的记录标识符从表中读取列。

● TBSCAN：通过直接从数据页中读取所有必要的数据来检索行。
● IXSCAN：用可选的启动/停止条件扫描表的索引，生成有序的行流。

4) 表扫描与索引扫描

如果没有创建合适的索引，或者索引扫描的成本更高一些，优化器通常就选择表扫描。当表十分小且索引集群率(index-clustering ratio)十分低，或查询需要很多表行时，索引扫描的成本可能更高。

前面的访问计划显示 SELECT 语句的总成本是 3420.41 *timeron*，这来自于表扫描操作符(TBSCAN)对 ACCOUNT 表进行的操作。因为该 SELECT 语句的结果集中只期望一行，所以表扫描在这里被认为是代价较高的操作。或者，可以尝试索引扫描，以便取得更好的性能。

在进行创建索引的工作之前，您可以运行 DB2 性能测试程序来获得性能基线，以便稍后进行比较。在本例中，10 秒中使用 12 000 个 4KB 页面缓冲池完成的事务数目是 598，如前面的图 11-18 所示。

在 ACCOUNT 表上创建索引，如下所示：

```
C:\> db2 connect to TESTDB
C:\> db2 describe table account
C:\> db2 "create index acct_id_inx on account (acct_id)"
```

用 RUNSTATS 命令更新目录统计数据，以便优化器可以考虑这个新创建的索引：

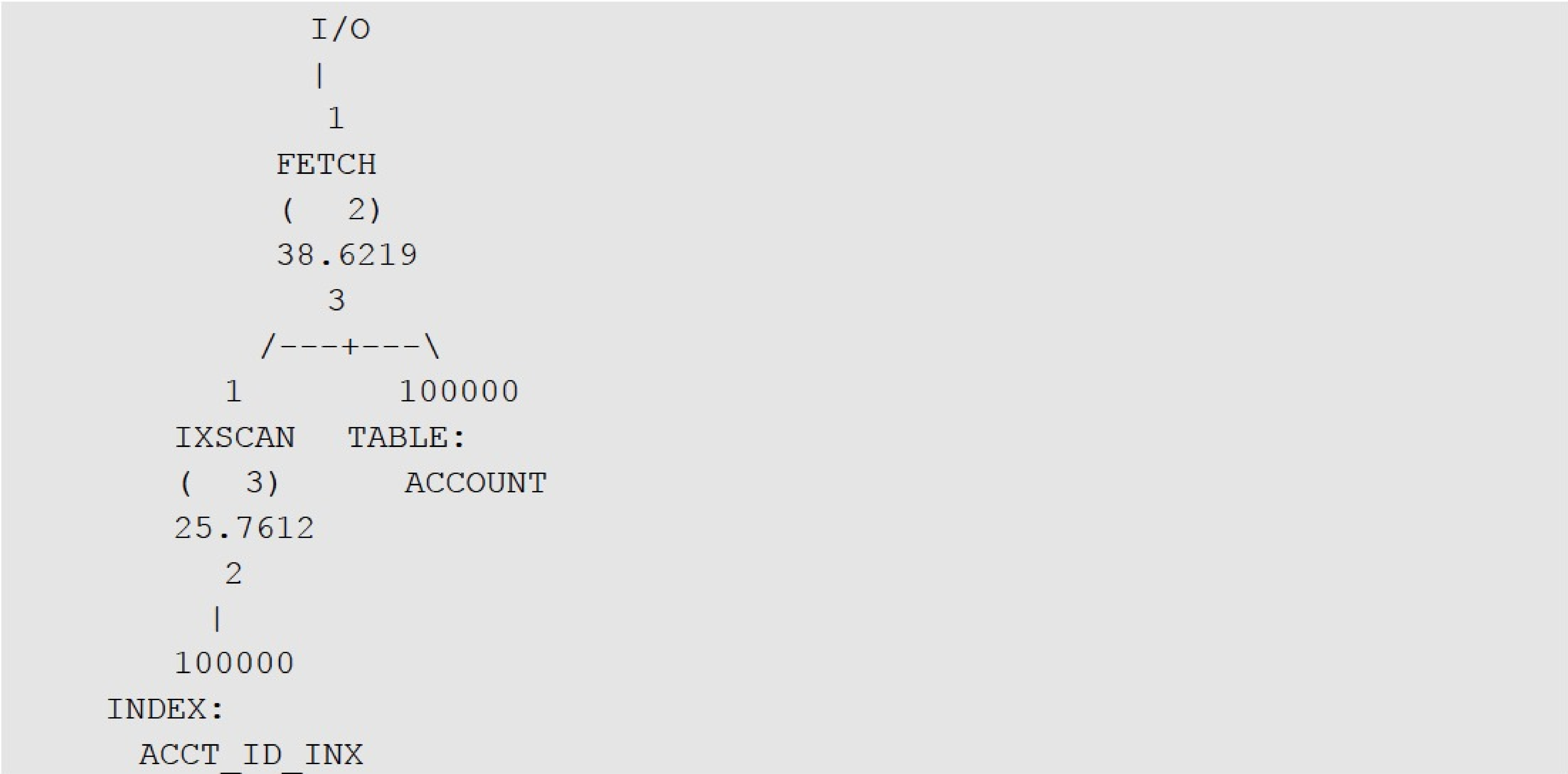
```
C:\> db2 runstats on table account with distribution and detailed indexes all
```

为了在 ACCOUNT 表上创建索引之后评估新的访问计划，要发出：

```
C:\> db2 set current explain mode explain
C:\> db2 -tvf select.ddl
C:\> db2 set current explain mode no
C:\> db2exfmt -d TESTDB -g TIC -w -1 -n C:\> -s C:\> -# 0 -o explain_index.out
```

下面展示了新的访问计划：

```
Access Plan:
-----------
        Total Cost:           38.6219
        Query Degree:          1
            Rows
           RETURN
           (1)
            Cost
```



```
             I/O
             |
              1
           FETCH
           (   2)
           38.6219
              3
          /---+---\
        1         100000
     IXSCAN   TABLE:
     (   3)      ACCOUNT
     25.7612
        2
        |
     100000
 INDEX:
   ACCT_ID_INX
```

通过在 ACCT_ID_INX 列上添加索引，新的访问计划显示了从 3420.41 到 38.6219 *timeron* 的大幅度成本减小。

图 11-20 展示了创建索引之后 DB2 性能测试程序的性能得到了极大提高，从 598 个事务提高到 1744 个事务。



图 11-20　使用 ACCOUNT 表上的索引扫描的性能结果

7. 实验总结

这个使用 Java 示例程序(DB2 性能测试程序)的调优练习介绍了 DB2 性能监控和调优的基础知识。您可以应用这些简单的一步步的性能调优示例来提高您自己的 DB2 数据库系统的性能。此外，您也有机会了解如何评估和分析访问计划，并修复“糟糕的查询”。